UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMATICA

TESIS DOCTORAL

Modelling and validation of cloud systems using model driven engineering, metamorphic and mutation testing

Modelado y validación de sistemas cloud utilizando ingeniería dirigida por modelos, testing metamórfico y de mutación

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Pablo Cerro Cañizares

Directores

Alberto Núñez Covarrubias
Juan de Lara Jaramillo

Madrid

UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

TESIS DOCTORAL

Modelado y validación de sistemas cloud utilizando ingeniería dirigida por modelos, testing metamórfico y de mutación. Modelling and validation of cloud systems using model driven engineering, metamorphic and mutation testing.

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Pablo Cerro Cañizares

DIRECTOR

Alberto Núñez Covarrubias y Juan de Lara Jaramillo

UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

TESIS DOCTORAL

Modelling and validation of cloud systems using model driven engineering, metamorphic and mutation testing

Modelado y validación de sistemas cloud utilizando ingeniería dirigida por modelos, testing metamórfico y de mutación

Autor

Pablo Cerro Cañizares

Directores

Alberto Núñez Covarrubias, Juan de Lara Jaramillo

Facultad de Informática
Universidad Complutense de Madrid

Madrid, 2020

Modelling and validation of cloud systems using model driven engineering, metamorphic and mutation testing

Modelado y validación de sistemas cloud utilizando ingeniería dirigida por modelos, testing metamórfico y de mutación

Thesis by

Pablo Cerro Cañizares

Advisors

Alberto Núñez Covarrubias, Juan de Lara Jaramillo

Facultad de Informática
Universidad Complutense de Madrid

Madrid, 2020

Modelling and validation of cloud systems using model driven engineering, metamorphic and mutation testing

Modelado y validación de sistemas cloud utilizando ingeniería dirigida por modelos, testing metamórfico y de mutación

Memoria que presenta para optar al título de Doctor en Informática

Pablo Cerro Cañizares

Dirigida por los profesores

Alberto Núñez Covarrubias, Juan de Lara Jaramillo

Facultad de Informática
Universidad Complutense de Madrid

Madrid, 2020

UNIVERSIDAD COMPLUTENSE MADRID

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD DE LA TESIS PRESENTADA PARA OBTENER EL TÍTULO DE DOCTOR

D./Dña. Pablo Cerro Cañizares,
estudiante en el Programa de Doctorado en Ingeniería Informática,
de la Facultad de Informática de la Universidad Complutense de Madrid, como autor/a de la tesis presentada para la obtención del título de Doctor y titulada:

Modelado y validación de sistemas cloud utilizando ingeniería dirigida por modelos, testing metamórfico y de mutación.

Modelling and validation of cloud systems using model driven engineering,metamorphic and mutation testing.

y dirigida por: Alberto Núñez Covarrubias y Juan de Lara Jaramillo

DECLARO QUE:

La tesis es una obra original que no infringe los derechos de propiedad intelectual ni los derechos de propiedad industrial u otros, de acuerdo con el ordenamiento jurídico vigente, en particular, la Ley de Propiedad Intelectual (R.D. legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual, modificado por la Ley 2/2019, de 1 de marzo, regularizando, aclarando y armonizando las disposiciones legales vigentes sobre la materia), en particular, las disposiciones referidas al derecho de cita.

Del mismo modo, asumo frente a la Universidad cualquier responsabilidad que pudiera derivarse de la autoría o falta de originalidad del contenido de la tesis presentada de conformidad con el ordenamiento jurídico vigente.

En Madrid, a 12 de diciembre de 2019

Fdo.: _____

Esta DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD debe ser insertada en la primera página de la tesis presentada para la obtención del título de Doctor.

This thesis has been developed within the Design and Testing of Reliable Systems research group of the Complutense University of Madrid (group number 910606 of the catalogue of groups recognised by the UCM) and has been supported by the following research projects:

- Spanish MINECO-FEDER (grant numbers DArDOS TIN2015-65845-C3-1-R, FAME RTI2018-093608-B-C31 and MASSIVE RTI2018-095255-B-I00)
- The Region of Madrid (grant numbers SICOMORo-CM ICE/3006 and FORTE-CM S2018/TCS-4314)
- The Universidad Complutense de Madrid - Santander Universidades grant (CT17/17-CT18/17)

Acknowledgements

Some people want it to happen, some wish it would happen, others make it happen.

Michael Jordan

This doctoral thesis is dedicated to my PhD advisors, my family, and everyone who supported me during this long voyage.

I would like to thank the rapporteurs of the PhD: Fatiha Zaidi and Rosa Filgueira.

Abstract

Stay far from timid only make moves
when you're heart's in it, and live the
phrase the sky's the limit.

The Notorious B.I.G.

Cloud systems are supported by complex infrastructures, which consist of a wide diversity of subsystems and components like storage, virtualisation and networking. The heterogeneous nature of these systems, their size, the high number of users that concurrently request services, and the virtualisation used to give the illusion of using dedicated machines, among other factors, hamper their validation. Unfortunately, it is not feasible to use conventional testing methods for checking the correctness of cloud systems.

The main goal of this thesis is to design methodologies and techniques for modelling and testing cloud systems. For this, the thesis combines two orthogonal techniques, Testing and Model Driven Engineering (in short, MDE) to model, validate and optimise cloud systems in a viable and effective way.

On the one hand, we use techniques based on formal testing to check the correctness of cloud systems. These techniques allow us to evaluate, from a precise definition of its behaviour, whether the systems act as expected. Specifically, we provide a methodology based on Metamorphic Testing (in short, MeT), which uses relevant properties to check the correctness of the system under study without the need to have implicit knowledge about it. Moreover we use MeT, in combination with Mutation Testing (in short, MuT) for analysing the effectiveness of these properties to detect anomalous behaviours of the system.

On the other hand, MDE-based techniques are applied to accurately represent through a meta-model the infrastructure of the cloud along with their underlying properties. In order to detect design issues and to provide solutions for misconfiguration of cloud systems, the thesis proposes a set of expert rules and a graphical language to facilitate the design of cloud systems.

It is important to emphasise that in this thesis non-functional aspects of the system, like energy consumption are studied in depth. Thus, the first

goal of the optimisation process is to define the meaning of an 'incorrect behaviour' of the system, since it is possible that, even if the correct test results are obtained through the comparison of its outputs, the analysed (non-functional) properties of the system under study do not reflect the expected behaviour. Hence, once the fault is located, valid alternatives to correct it and to optimise the property under study in the testing process are proposed. In order to locate these alternatives, this thesis propose techniques inspired by Evolutionary Algorithms (in short, EAs), which focus on an adaptive global search for possible solutions. This technique provides almost optimal solutions to complex optimisation problems, where the execution time represents a significant limitation.

Keywords: Metamorphic testing, mutation testing, model driven engineering, cloud computing, evolutionary algorithms.

Resumen

Ningún sueño es demasiado grande,
el límite es el cielo.

The Notorious B.I.G.

Los sistemas cloud están formados por infraestructuras complejas, donde existe una gran diversidad de subsistemas y componentes, tales como el almacenamiento, la virtualización y las redes. La heterogeneidad de estos sistemas, su amplitud, el elevado número de usuarios que solicitan servicios de forma simultánea y la virtualización utilizada para ofrecer la ilusión de utilizar máquinas dedicadas, entre otros factores, dificultan su validación. Desafortunadamente, utilizar métodos de Testing convencionales para comprobar la correción de los sistemas cloud no es factible.

El objetivo principal de esta tesis es diseñar metodologías y técnicas para modelar y testear sistemas cloud. Para ello, esta tesis combina dos líneas ortogonales – Ingeniería Dirigida por Modelos o MDE (por las siglas de Model Driven Engineering) y Testing – apoyadas sobre una base formal, para desarrollar metodologías y técnicas que permitan optimizar sistemas cloud de forma viable y eficiente.

Por un lado, utilizamos técnicas basadas en Testing formal para comprobar la corrección de los sistemas cloud. Estas técnicas nos permiten evaluar, a partir de una definición precisa de su comportamiento, si los sistemas actúan como se espera. Específicamente, proporcionamos una metodología basada en Testing Metamórfico o MeT (por las siglas de Metamorphic Testing), que utiliza propiedades relevantes para comprobar la corrección del sistema bajo estudio sin necesidad de tener un conocimiento implícito del mismo. Además, utilizamos MeT, en combinación con Testing de Mutación o MuT (por las siglas de Mutation Testing) para analizar la eficacia de estas propiedades y detectar comportamientos anómalos del sistema.

Por otro lado, se aplican técnicas basadas en MDE para representar con precisión, a través de un meta-modelo, la infraestructura de los sistemas cloud junto con sus propiedades subyacentes. Con el fin de detectar problemas de diseño y proporcionar soluciones a errores de configuración de los sistemas cloud, se han incluido un conjunto de reglas de experto y un

lenguaje gráfico para facilitar el diseño de los mismos.

Es importante remarcar que en esta tesis se profundizará en el estudio de los aspectos no funcionales del sistema, tales como el consumo energético. Así, el proceso de optimización tendrá como primer objetivo definir qué es un 'comportamiento incorrecto' del sistema, ya que es posible que, aún obteniendo los resultados correctos de los tests mediante la comparación de sus salidas, las propiedades analizadas del sistema bajo estudio no reflejen el comportamiento esperado.

De esta forma, una vez localizada la anomalía, propondremos un conjunto de alternativas válidas que la subsanen, de forma que se optimice – en la medida de la posible – la propiedad bajo estudio en el proceso de testeo. Para localizar estas alternativas se utilizarán técnicas inspiradas en algoritmos evolutivos o EAs (por las siglas de Evolutionary Algorithms), los cuales se centran en una búsqueda global adaptativa sobre el espacio de posibles soluciones, proporcionando soluciones casi óptimas a problemas de optimización complejos, donde el tiempo de ejecución representa una limitación significativa.

Palabras clave: Testing metamórfico, testing de mutación, ingeniería dirigida por modelos, sistemas cloud, algoritmos evolutivos.

Contents

List of Figures

List of Tables

Abbreviations

ACO Ant Colony Optimisation.

BCO Bee Colony Optimisation.

CDT C/C++ Development Tooling parser.

CCM Cloud Chromosome.

CCP Close Page Policy.

DSL Domain Specific Language.

EA Evolutionary Algorithm.

EC Energy Consumption.

EM Evaluation Module.

EMF Eclipse Modelling Framework.

EVL Epsilon Validation Language.

FDB Factual Database.

GMF Graphical Modelling Framework.

FCFS First Come First Serve.

GPML General Purpose Modelling Language.

GA Genetic Algorithm.

GP Genetic Programming.

GUI Graphical User Interface.

HPC High Performance Computing.

HPMA High Performance Memory Access.

IE Inference Engine.

KB Knowledge Base.

MI Million of instructions.

MeT Metamorphic Testing.

MOF Meta-Object Facility.

MuT Mutation Testing.

MR Metamorphic Relation.

NIST National Institute of Standards and Technology.

OCL Object Constraint Language.

OMG Object Management Group.

RDAF Request Density Aware Fair Memory.

SLA Service-Level Agreement.

SOA Software Oriented Architecture.

SP Simulation Platform.

SUT System Under Test.

UML Unified Modelling Language.

USIMM Utah SImulated Memory Module.

TC Test Case.

TCE Trivial Compiler Equivalence.

TS Test Suite.

VM Virtual Machine.

Part I

Summary of the Research

Chapter 1

Introduction

Had a dream, I was King.
I woke up, still King.

Marshall Mathers III

This chapter presents the main contributions of this thesis and the challenges faced to accomplish them. Section 1.1 presents introductory concepts and the motivation and challenges found in the field of modelling and testing cloud systems. Section 1.2 describes the main objectives of this thesis. Section 1.3 explains in detail the main contributions of this thesis. Finally, the structure of this document is summarised in Section 1.6.

1.1 Presentation and motivation

In recent years cloud computing has gained significant attention due to its flexible and on-demand computing infrastructure. This interest has a significant impact in both the IT industry and the research community. On the one hand, leading companies such as IBM, Microsoft, Google and Amazon have spent valuable resources on cloud computing [179]. On the other hand, researchers are continuously developing tools and techniques to improve this emerging technology [18, 115]. However, although cloud computing is considered one of the most cost-effective solution for final users, several factors have to be managed when providing a system with features such as 24/7 availability, worldwide utilisation, and easy access for every user.

One of the main problems that we have to overcome when developing cloud computing systems is to ensure that the behaviour of the system is consistent with expectations, which includes factors such as performance and user management. Moreover, additional factors make cloud systems particularly complex. First, the large size of cloud systems hampers their analysis and study. Second, the resources of the cloud provided to end-users

are *virtual*, and this complicates its analysis since different Virtual Machines (in short, VMs) can be hosted on a single physical machine, sharing the same resource among different users. Finally, we cannot neglect the vast number of users that concurrently use a cloud system. Currently, *testing* is the most widely used technique to validate the correctness of computer systems [6, 125]. If we start the development of a system from a formal model, then testing can be used to perform more rigorous analysis [75]. However, the development of formal testing methodologies for checking cloud systems entails a significant challenge [43].

In testing, it is normal to use an *oracle* that checks whether the behaviour observed during testing, with a given test case, is allowed/acceptable. An oracle can be a realisation of a formal specification of the system or a set of properties that the system has to fulfil. However, in some situations, an oracle is not available or is computationally too expensive to be applied. This fact hampers distinguishing the correct behaviour from potentially incorrect behaviours of a system and is known as the *oracle problem* [14, 180]. Additionally, testing complex systems, like clouds, requires generating and executing large test suites to completely check the behaviour of the system. In general, executing all possible test cases is not possible due to computer power and time constraints and, therefore, an appropriate subset of test cases must be selected for determining the correctness of the system. This is known as the *reliable test set problem* [49, 73]. These problems specially arise in cloud computing systems, where an oracle indicating whether the design of a cloud system is correct is rarely available and quality test suites are required to check the critical parts of the system.

The main objective of this thesis is the study, research and development of testing and modelling techniques to check the correctness of cloud systems. In order to accomplish this goal, the thesis can be divided in three main parts.

The first part deals with the design and development of a formal methodology for testing different aspects of the cloud. The methodology integrates simulation – to represent the behaviour of cloud computing systems – with testing methods for checking the correctness of the modelled cloud systems. The main goal is to provide a methodology, supported by a tool, such that users can model both the software and hardware parts of cloud systems, design new cloud system models and automatically test these models using a cost-effective approach that considers both functional and non-functional aspects of the cloud. In order to alleviate both the *reliable test set problem* and the *oracle problem* to check cloud systems, we use techniques inspired by metamorphic testing (in short, MeT). MeT uses expected properties of the target functions to alleviate these problems [48, 49, 134]. These properties relate multiple test-inputs/observed-outputs obtained from the tested system using metamorphic relations (in short, MRs). The methodology in-

tegrates simulation techniques to alleviate the inconveniences associated to this kind of techniques: reducing costs by avoiding renting real systems, providing more flexibility in comparison with private systems and improving the scalability.

The second part deals with the integration of mutation testing (in short, MuT) techniques for improving the methodology proposed in the previous part [56, 71]. The main goal is to apply MuT to analyse the effectiveness of the MRs in finding errors. This testing technique is used to automatically inject faults in the software under study. Thus, we investigate both, the software part of cloud systems and the applications that are executed on them. Moreover, several optimisations have been included to parallelise the MuT process, and alleviate, in this way, the high computational cost associated with this testing technique.

The third part deals with the formal modelling of cloud systems. Therefore, it is necessary to use mechanisms that allow modelling, with a high degree of fidelity, the behaviour of cloud systems, their functionality, performance and energy consumption, as well as the services deployed on them. In order to achieve this objective, we use an Model Driven Engineering (in short, MDE) approach [23, 159]. This technique allows us to improve the flexibility and quality of the models with respect to classical modelling methods. Hence, the inclusion of MDE techniques aid users to easily design and manage cloud systems.

1.1.1 Metamorphic Testing

Traditional testing techniques require checking the conformance between the input(s) and the output(s) of the system under study. Schematically, let $\mathcal{S}$ be a system, I the input domain, O the output domain, and TS a test selection strategy. Let $\mathcal{T} = \{t_1, t_2, \ldots, t_n\} \subseteq I$ be the set of tests generated by using TS. When these tests are sequentially applied to the system $\mathcal{S}$ (which can be modelled as a function $S : I \rightarrow O$) we obtain a sequence of outputs $\mathcal{S}(t_1), \mathcal{S}(t_2), ..., \mathcal{S}(t_n)$. Given an oracle f, an error is found in $\mathcal{S}$ if there exists $t_i \in \mathcal{T}$ such that $\mathcal{S}(t_i) \neq f(t_i)$.

However, a complete oracle f, able to exactly characterise the expected output of a test, is challenging in many domains, including cloud computing. In order to alleviate this problem, we propose using MeT techniques [48, 58, 112]. The main difference between traditional testing techniques and MeT lies in the comparison of the obtained outputs. Hence, while traditional techniques compare the output of each individual test case with the one obtained from the oracle, MeT checks the relation between multiple test inputs and their outputs, and therefore an additional oracle is not needed.

MeT uses expected properties of the target system relating multiple test inputs with the corresponding outputs obtained from the system under test.

These properties are formulated as metamorphic relations (in short, MR). A MR is a property of the analysed system that involves multiple inputs and their outputs. We represent an MR as a tuple (MR_i, MR_o), where MR_i refers to the relation between the source test case and the follow-up test case, and MR_o refers to the relation that must be fulfilled by the outputs obtained from the source test case and the follow-up test case.

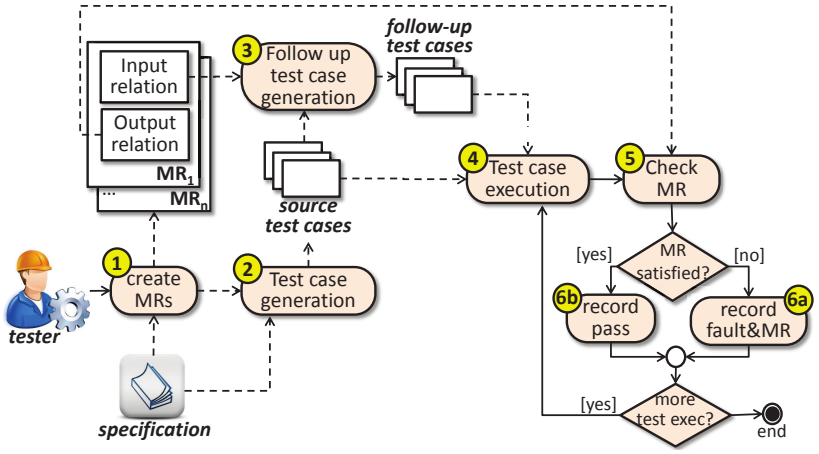


Figure 1.1: Scheme of the MeT process.

Figure 1.1 illustrates the MeT process. Initially (activity with label ①), the tester must build a repository of MRs, which act as an oracle to check whether the outputs returned by the system under test are the expected ones. These MRs must be designed according to the specification of the system under test. Please note that the tester must be able to interpret the specification of the system under test, if it is available, to properly build suitable MRs. This task is especially challenging when the system under study is complex.

Next, for each MR in the repository of MRs, the tester must build a test suite consisting of a collection of source test cases. These test cases must be generated considering the specification of the system under test (label ② in Figure 1.1). However, any traditional testing technique, like random testing [53], can be used to create each test suite. Similarly, in the next step (label ③), for each previously generated source test suite in ②, a new follow-up test suite, containing the same number of test cases, is built. Thus, follow-up test cases are generated by using both the source test cases and the input relation of MR, that is, MR_i.

Next, each test case generated in the previous steps is executed against the system under study (label ④). When the execution of all the test cases finishes, the MRs are used to check the obtained outputs (label ⑤). In order to accomplish this task, these MRs are chosen one by one from the repository of MRs created in ①. Hence, for each MR, the source and follow-

up test cases are used to check whether their outputs satisfy the relation given by MR_o. If the relation is not satisfied, an error has been found, and the corresponding fault and the violated MR are stored to analyse the issue (label 6a). On the contrary, if the *MR* is fulfilled, the statistics are updated, increasing the number of test cases that satisfy this relation (label 6b). Once all the test cases are checked against the *MR*, the next MR is chosen. This process is repeated until all the MRs are processed.

1.1.2 Mutation Testing

MuT is a fault-based testing technique that provides a testing criterion called the mutation adequacy score. The main goal is to measure the effectiveness of a test suite in terms of its ability to detect faults. Hence, MuT is an alternative to other approaches to evaluate tests suites, such as statement/branch coverage [7].

The faults are injected into the code using *mutation operators*, which correspond to rules for transforming the syntax of the language. The idea is to create copies of the original program, where each copy presents some syntactic modification. The objective is to determine the number of program variations, called *mutants*, which behave differently from the original program with respect to a test suite. When a mutant behaves differently from the original program, for some test `t`, it is said that `t` *kills* the mutant, otherwise, it is said that the mutant is *alive*.

Let us consider the example of the Table 1.1, where the original program is mutated in such a way that the condition $a > 0 \wedge b > 0$ is replaced by $a > 0 \vee b > 0$. This modification generates a mutant. In order to kill this mutant, it is necessary to create a test that causes a different result in the execution of the mutant with respect to the original program. If we apply the test $a = -3, b = 5$, we obtain different results from the original program, and therefore, the mutant is killed. However, the test $a = 3, b = 5$ will not detect the failure because we obtain the same result in both the original and the mutant. Hence, the mutants allow evaluating how good the tests are, that is, the more mutants killed by the selected set of tests, the higher the quality of the tests.

Some mutants, called *equivalent* mutants, present the same behaviour as the original program for any input and, consequently, cannot be killed by any test. The detection of *equivalent* mutants is an undecidable problem, so they must be detected manually, which means a high cost in the application of this technique [24]. In order to mitigate this problem, there exist several techniques, such as those proposed in the work of Offutt and Craft [131], where algorithms are detailed to determine different classes of equivalent mutants. These algorithms are based on data flow analysis and compiler optimisation techniques. In another relevant work in this field, Robert Hierons and Mark Harman present the use of *slicing programming* techniques

Program p	Program p'
...	...
if(a>0 **and** b>0)	if(a>0 **or** b>0)
return 1	return 1
else	else
return 2	return 2
...	...

Table 1.1: Example of mutated code.

applied to the detection of equivalent mutants, which allow reducing their generation cost [76].

MuT is based on two hypotheses related to the nature of errors in programs. On the one hand, it is considered that programmers tend to develop programs that are correct. This hypothesis, known as *competent programmer hypothesis* [126], states that bugs in programs are small syntactic errors that, in most cases, could easily be corrected. Therefore, mutants simulate possible effects of actual bugs. Thus, if a set of tests is good at detecting errors in the mutants, it will also be good at detecting errors in the original program. The second hypothesis, known as *coupling effect hypothesis* [56], states that complex errors are correlated with simple errors so that a set of input tests that detects all simple faults in a program will detect a high percentage of complex faults. There are numerous studies that consider the validity of this hypothesis, both theoretical [56, 130] and practical [111].

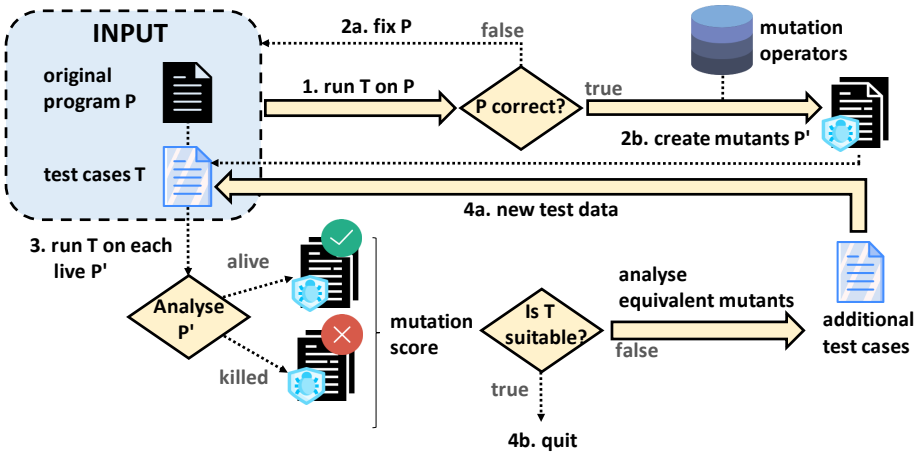


Figure 1.2: General scheme of the MuT process.

Figure 1.2 presents the general operation scheme of MuT. Given a set of mutants, generated from applying a set of mutation operators over a

program, and a test suite, the adequacy of the test suite is calculated by analysing its ability to detect failures. First, the test suite must be applied to the original program to ensure its correct behaviour ①. If the results are incorrect, the original program must be corrected ②ⓐ. On the contrary, the mutation operators are applied to the original program for the generation of all possible mutants ②ⓑ. Each mutant will be executed using the provided test suite ③. At the end of this process, the mutation score will be calculated, indicating the percentage of non-equivalent mutants that have been killed by the test suite. The goal is to achieve a mutation score of 100%, i.e. all non-equivalent mutants have been killed. Alive mutants indicate a lack of adequacy of the test suite to detect potential failures in the program. Therefore, taking the example of the Table 1.1, for a test where $a = 8, b = 7$ both p and p' would have the same result. Hence, p' would be considered as an alive mutant. In this case, the user must analyse whether there exist equivalent mutants and discard them. Then, if after discarding the equivalent mutants there still exist alive mutants, it is necessary to add additional tests to the initial test suite and check that the original program satisfies them, to try to kill the mutants that remain alive ④ⓐ. This process should be repeated until the user is satisfied with the mutation score obtained ④ⓑ.

1.1.3 Model Driven Engineering

MDE is a software development methodology based on the use of different abstract representations of systems known as *models*. MDE technologies combine modelling languages, which allow specifying models, model transformations to increase the automation of model manipulations and code generation to obtain executable code from models.

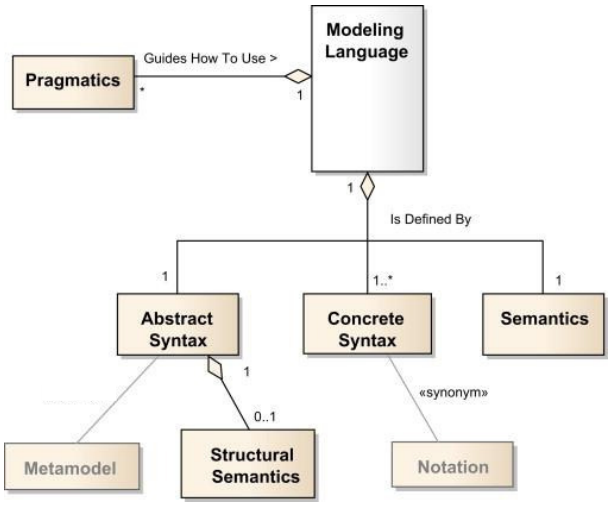


Figure 1.3: Definition of modelling languages (from [9].)

Modelling languages can be classified, depending on their purpose, in two different groups. The *General Purpose Modelling Languages* (in short, GPMLs) can be applied on a wide spectrum of fields, providing a high number of generic constructs and notations. Thus, the goal of GPMLs is to enable the design of any kind of system with them. The Unified Modelling Language (in short, UML) [1, 154] is the most extended GPML in both industry and research. However, these languages rarely capture particular domain concepts. In contrast to GPMLs, *Domain Specific Languages* (in short, DSLs) are tailored to a specific domain [97, 176]. DSLs provide a high-level abstraction that is achieved through the use of domain concepts, which improves the readability, understanding and the communication between developers and domain knowledge experts. Some studies show that DSLs can improve the productivity of the construction process, and the reliability, maintainability and portability of the resulting systems [2, 74]. Many examples of DSLs can be found in the literature, including, HTML [171], LATEX [108] and MATLAB [117].

A modelling language is made of the four elements depicted in Figure 1.3: an abstract syntax, a concrete syntax, a description of the semantics and the pragmatics of the language.

- The abstract syntax represents the main concepts, abstractions, and their underlying relations of the application domain. It is strongly associated with the domain knowledge. The abstract syntax of a language is often described by a *meta-model*. A meta-model can be considered as a model of a class of models [164].

- The concrete syntax corresponds to the specific representation of the language. It can be textual or graphical. This component affects the user experience in terms of effectiveness.

- The semantics refer to the mathematical model that reflects the computational behaviour of syntactically valid models of a language. The static semantics defines structural properties, that is, well-formedness constraints that models should obey. The dynamic semantics describes the execution behaviour of the model. There exist different techniques to formally define the semantics of a language, like operational, translational, and denotational [72].

- Pragmatics concern the meaning and interpretation of the language depending on the context. In some instances, several languages include tutorials, guidelines, and examples in their own definition to use the language in a proper way.

Figure 1.4 illustrates the general scheme of the four-layer architecture used as a reference in the MDE context [120]. At the top level of abstraction

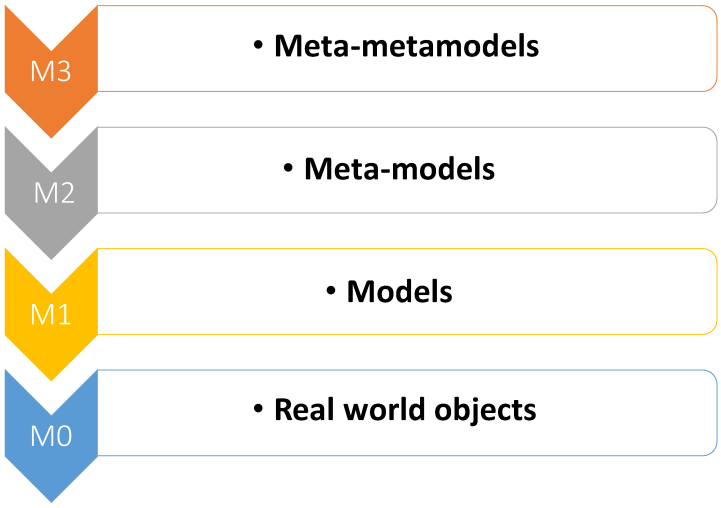


Figure 1.4: Four-layer architecture in MDE.

(*M3*) we can find the meta-metamodel layer, which is used to define meta-models. The Meta-Object Facility (in short, MOF) is a standard hosted by the Object Management Group (in short, OMG) that is commonly used for defining meta-models. At the next level down in the hierarchy (*M2*), the meta-models specify models to define the structure of a modelling language. In the layer below (*M1*) we have the models, that are instances of the meta-models and are designed according to the users requirements. Finally, in the last layer (*M0*) we find the real-world objects, which are modelled by the upper layers. While MOF itself is also expressed as a model, it is often called a meta-metamodel.

1.2 Objectives

The main objective of this thesis is to combine testing and modelling techniques to check the correctness of cloud systems. This goal can be divided in three parts:

- Providing a methodology to analyse the correctness of the cloud systems in an automatic, simple and viable way. The basic idea is to provide users, who lack specific knowledge in the area, with a – tool supported – framework that supports performing MeT in cloud systems, without previous knowledge about the infrastructure of the system under study.
- Designing MuT techniques for testing cloud environments. In essence, these techniques are focused on improving the methodology proposed in the previous objective. Firstly, providing a mechanism to properly

analyse the distributed applications that are executed in cloud systems. That is, generating test cases for testing the applications under study. Secondly, generating faulty clouds, and seeding bugs in the software parts of the cloud systems, such as schedulers, cloud providers and planning policies, among others. Finally, providing a set of optimisations – inspired by High Performance Computing (in short, HPC) techniques – for reducing the computational cost of MuT.

- Providing a formal model of the cloud using MDE. The modelling process is focused on representing the behaviour of the cloud accurately. For this reason, the modelling process starts from the most essential components, such as CPU, memory and storage systems, to model more complex elements, such as computing and storage nodes, communications networks and virtualisation of hardware resources.

1.3 Contributions

The main contributions of this thesis are categorised in different lines: MeT, MuT and MDE.

The principle advances performed in the field of **Metamorphic Testing** are described in Chapter 3. In summary, the contributions are described below:

- Designing a methodology for optimising the energy consumption of cloud infrastructures. In order to check and optimise the energy consumption of cloud systems, we propose a novel approach that combines MeT and EAs. The idea is to use an EA to evolve cloud systems efficiently. This is achieved by using MRs, which guide the search, for finding an optimised cloud. This way, each new offspring of individuals (clouds) are generated using the constraints defined in the MRs. Thus, the proposed EA not only reduces the search space but improves the overall efficiency. This work has been presented in [38].

- Designing a methodology for checking the correctness of memory systems, which are considered a key component of the physical machines that support the cloud systems, using simulation and MeT. The main goal of this contribution is two-fold. Firstly, providing a method to automate the testing process of memory systems. Secondly, implementing a novel expert system focusing on increasing the overall performance of the testing process. In contrast to conventional expert systems, the proposed system includes a factual database containing the results of previous simulations, and a simulation platform for computing the behaviour of memory systems. The knowledge of the expert is represented in the form of MRs, which are properties of the analysed system

involving multiple inputs and their outputs. The developed expert system was able to detect over 99% of the critical injected faults, hence obtaining very promising results, and outperforming other standard techniques like random testing. This work has been presented in [30].

In the case of the progress achieved in the field of **Mutation Testing**, our contributions can be divided in two main lines. The first one consists of a set of optimisation strategies to improve the performance of the mutation process, which are described in detail in Section 4.3. The second line provides the basis for applying MuT techniques on highly distributed systems is presented in on Section 4.2. These contributions are summarised as follows:

- Designing a scalable, dynamic, and HPC-oriented framework, based on embarrassingly parallel computation ideas to reduce the execution time associated to the classical MuT scheme. This contribution has been presented in [31].
- Providing different optimisations to improve the overall MuT process. In essence, these techniques have been included in the previous framework to reduce the computational cost associated to this testing process. During the experimental phase, we identified some drawbacks that prevented the proposal to exploit the resource usage in HPC systems. Hence, we propose an optimisation consisting of 4 different strategies to alleviate this issue: parallelising the execution of the TS over the original application, sorting the TS using the execution time of each test case, enhancing the test case distribution and grouping cloned and equivalent mutants. This contribution has been presented in [37].
- Applying MuT in highly scalable distributed systems. For this, we propose the design of a framework to analyse the effectiveness of test suites for detecting errors in distributed applications. Thus, we propose several mutation operators focused on simulation. These operators represent different errors made by competent programmers over these platforms. The proposal has been implemented in a tool called MuTomVo, which integrates the proposed MuT framework in a simulation tool. In order to support the feasibility of the proposal, we have conducted an experimental phase over three applications running in different distributed systems: a client/server model, an intensive computation application and a scientific pipeline. This contribution has been presented in [39].

Regarding to the field of **Model Driven Engineering**, several techniques based on MDE principles have been analysed to model cloud systems. The results of this study are described at Section 5.

- Providing a model-based framework for designing and analysing the data-centre configurations that support cloud systems. The framework allows the user to easily design a complete data-centre. Also, it relies on a library of expert knowledge rules to detect a wide range of misconfigurations and suggest improvements in the design. Then, we support the simulation of the data-centre configuration to assess properties like scalability, and detect possible bottlenecks. This advance has been presented in [36].

Figure 1.5 illustrates the contributions of this thesis and the relations between them. The top of the figure shows the three main research lines of this thesis, that is, MeT, MuT and MDE, which use different techniques such as EAs, simulation and structural rules, to analyse the software and hardware parts of the cloud.

In this way, cloud systems can be analysed at three levels:

- Combining MeT with EAs and simulation for checking non-functional properties of the cloud and providing optimisations of its architectural design.

- Combining MuT with simulation for analysing the software parts of the cloud, like resource application policies, and the applications executed in it.

- Combining MDE with simulation and structural rules for modelling, checking the correctness and optimising data-centre configurations that support cloud systems.

It should be noted that MeT and MuT are complementary testing techniques. MuT is used to analyse the correctness of the MRs, providing a suitable and appropriate test suites, and improving the overall performance of MeT. In addition, the cloud models designed using MDE can be tested using MeT.

1.4 Technical contributions

In addition to the previously mentioned fundamental contributions, this thesis also provides the following technical contributions:

- The development of a framework for analysing and improving energy aware cloud systems. This framework implements a catalogue of MRs and supports the proposed methodology to optimise cloud systems using MeT and EAs.

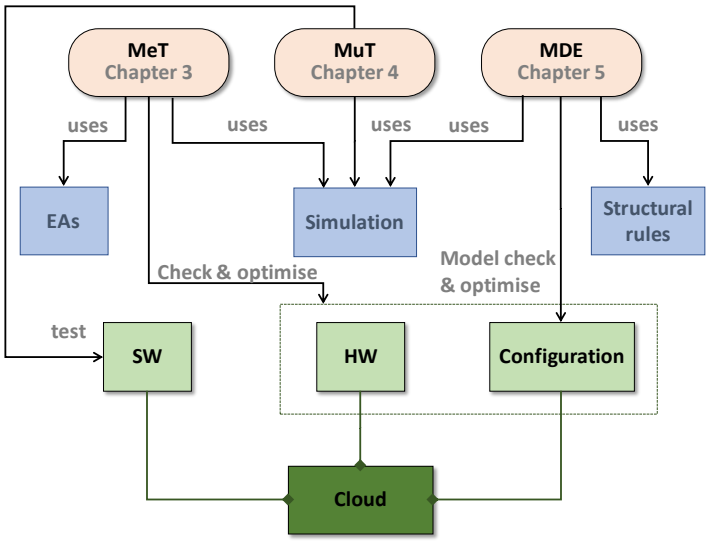


Figure 1.5: Integration scheme of the contributions provided in this thesis.

- The development of a framework for checking the correctness of memory systems, which includes a catalogue of MRs and its core is supported by an expert system.

- The development of a framework, called *Malone*, for improving the performance of MuT. *Malone* implements the complete set of optimisations focused on reducing the overall time of MuT proposed in this thesis.

- The development of *MuTomVo*, a MuT framework for simulated cloud and HPC environments. *MuTomVo* implements a set of mutation operators, two different test case generators, a generic mutation engine and a graphical GUI for executing analysing the experimental setting required by the user.

- The development of an Eclipse plugin called *MAGICIAN*, a MDE-based framework for designing and optimising cloud data-centres. *MAGICIAN* includes a graphical DSL for easily design data-centre infrastructures and a set of expert rules that help to find and fix design misconfigurations.

1.5 Research visits

During the realisation of this thesis, an external research stay was performed in collaboration with the EPCC of the University of Edinburgh, Scotland,

supervised by Dr. David Henty. This stay was performed in that University for a period of three months, from 23^{rd} july to 25^{th} october, 2018.

1.6 Summary

This thesis follows the format of compendium of publications according to the regulations in force in the Complutense University of Madrid and is divided into two parts. Part I summarises the results presented in this thesis. More concretely, the first chapter of Part I is an introduction and includes the objectives of the thesis and a summary of the contributions. The rest of the chapters are summarised as follows:

- Chapter 2 introduces preliminary knowledge to understand the content of the thesis. Initially, cloud computing is introduced in Section 2.1, simulation basics are described in Section 2.2 and basic notions of EA and MDE are introduced in Section 2.3 and 2.4, respectively.
- Chapter 3 presents the methodology for testing cloud systems using MeT and simulation. Initially, a summary of the existing works in the state of the art related with both MeT is provided in Section 3.1. Then, a description of the formal methodology for testing cloud systems based on MeT is presented in Section 3.2. Finally, a description of the architecture and testing procedure of the proposed expert system for analysing memory systems is introduced in Section 3.3.
- Chapter 4 describes the proposed MuT techniques for analysing cloud systems. Section 4.1 summarises the state of the art of parallel MuT techniques and proposals for testing distributed systems. The MuT framework for analysing cloud and HPC environments is described in Section 4.2. The advances achieved in the field of parallel MuT are presented in Section 4.3.
- Chapter 5 presents the techniques for modelling cloud systems. Section 5.1 summarises the principal advances existing in the state of the art related with the MDE field. Then, the meta-model architecture to describe all the components of a data-centre that supports a cloud system, and the graphical concrete syntax are provided in Section 5.2. The expert rules that aids the user to fix possible design errors are described in Section 5.3. Finally, the description of the GUI and the supporting technologies of the graphical language are provided in Section 5.4.
- Chapter 6 finishes with the conclusions and prospects for future work.

Part II presents the publications relevant to this thesis as they were originally published.

Chapter 2

Preliminaries

My mother taught me three things:
respect, knowledge, search for knowledge.
It's an eternal journey.

Tupac Shakur

This chapter presents a brief introduction of the background related to this thesis. Section 2.1 provides some notions about cloud computing, including its definition, management and system infrastructure concepts. Section 2.2 introduces the main features of simulation of cloud systems. In this section a thorough comparative study of the current cloud simulation platforms is included. Section 2.3 presents an introduction to evolutionary algorithms. Finally, Section 2.4 presents concepts and definitions of some of the fundamental parts of MDE.

2.1 Cloud computing

While the concept of cloud computing was formulated in 1997 [118], it only became a reality in 2007, showing a noticeable adoption by several leading companies like Google, IBM, Microsoft, and Amazon. Although there are currently several definitions of cloud computing, we adopt the one provided by the U.S. National Institute of Standards and Technology (in short, NIST), as it provides a formal description including key elements used in the cloud computing community [119]:

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

The underlying idea behind cloud computing consists of providing the

illusion of infinite computing resources available on demand. This way, *the cloud* is an autonomous system that is managed transparently to users, providing a unified platform containing software, hardware and data services.

According to the cloud deployment model [119], each cloud system can be categorised as *public*, *private*, *community* and *hybrid*. Public clouds are used by the general public cloud consumers. It is important to remark that the cloud service provider has the full ownership of the public cloud with its own policy, value, profit, cost, and charging model. This kind of cloud allows users to pay only for the capacity that their applications actually need, also known as *pay-as-you-go* model, and the cloud infrastructure is provisioned for exclusive use by a single organisation comprising multiple consumers. Public clouds do not allow the low-level architecture to be configured for experiments, and moreover, present significant variations in the overall performance depending on which machines the experiments are executed [158]. Amazon EC2 is a good example of public cloud [5].

Different from public clouds, private ones are operated solely within a single organisation. In general terms, the size of a private cloud is typically smaller than the size of a public one. One of the main motivations to setup a private cloud lies on security concerns. Since the organisation has full control over both data and infrastructure configurations, private clouds have become an option for many companies. Private clouds can be managed by the owner organisation itself or by a third party. In community clouds, the cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organisations that have shared concerns. It may be owned, managed, and operated by one or more of the organisations in the community, a third party, or some combination of them. Finally, hybrid clouds consist of a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities but are bound together by standardised or proprietary technology that enables data and application portability.

Managing a cloud system is a complex task. These systems usually consist of a huge quantity of physical nodes, whose resources are provided to the users for supporting their computing needs. At the same time, these resources are virtualised to maximise the use of the computational resources of the cloud. The proper operation of these mechanisms is a crucial aspect concerning both the correct behaviour and the overall system performance. To this end, the cloud provider and the hypervisor are considered essential in cloud environments.

The data-centre represents the physical resources that supports the cloud system [93]. These resources can be grouped into three categories: computing nodes, storage nodes and communication networks. The physical machine represents a computational node where the Virtual Machines (in short, VMs) requested by users are deployed, which consists of 4 subsystems: the CPU

system, the memory system, the storage system and network system. A VM is an emulated computer system that is created by using virtualisation. VMs are intended to provide users with the functionality of a complete computing node. The virtualisation consists of mapping virtual resources to real resources.

In general terms, the main tasks of a cloud provider consist of attending user requests, locating the resources requested by the user among the associated data-centres, and generating an answer for these requests. Specifically, the cloud provider is in charge of four main tasks. The first task is to manage the VMs of the cloud system, the second one is to manage the list of jobs submitted by the users, the third one is to schedule these jobs to be executed in the VM instances, and finally, to define cost policies for each VM instance type. Additionally, the cloud provider creates the illusion of managing infinite resources by hiding deployment details, even if the cloud owns several data-centres that are located in different geographical locations.

The hypervisor is considered a key part of a cloud. Basically, this module is in charge of handling hardware resources to host the VMs where the jobs defined by users are executed. In order to accomplish this task, the hypervisor can be fully configured by integrating customised brokering policies.

2.2 Simulation of cloud systems

Simulation is a well-known technique that has been used during the last few decades in several fields [20, 34, 39, 78]. In this thesis, simulation is considered as a key element, since it is used to simulate the cloud systems designed and analysed in the contributions [30, 32, 34, 36, 38, 39].

Robert E. Shannon defines simulation [165] as *the process of designing a model of a real system and conducting experiments with this model for the purpose of understanding the behaviour of the system and/or evaluating various strategies for the operation of the system.* In computer science, simulation is the technique of representing the real world by a computer program, which may imitate both the internal processes and the results of the system being simulated. Currently, the research community has adopted simulation as a widely used technique for analysing cloud systems [4, 63]. Among others, some of the most relevant advantages provided by simulation are described as follows:

- Executing experiments is cheap. The majority of the simulators has a GPL license [25], which does not require an investment for using the required tools. Moreover, simulations can be executed in single regular computers or, if available, in small commodity clusters for increasing the overall performance.

- Simulation does not require specific hardware for launching experi-

ments. Thus, the underlying architecture of the modelled system is not required and, therefore, any computer system can be used to execute the simulations.

- Simulation allows a high level of flexibility for performing experiments. A modelled system can be easily customised by modifying configuration parameters. In general, this is achieved by editing text files or setting parameters in a GUI. However, experimentation with real systems may require making changes in the hardware, which implies more time and effort.
- Experiments can be easily controlled and repeated in simulated environments.
- Models can be easily scaled, which consists of setting up the parameters involved with the size of the target system. On the contrary, scaling a real architecture is an expensive and time-consuming task.
- Simulators and models can be easily shared with other researchers.

However, the use of simulation does entail some drawbacks. Some simulation platforms do not provide real data since we only simulate the performance of a cloud system during a given experiment, in contrast to executing the experiment in a cloud system. In addition, not all simulators are focused on analysing and modelling the same parts of cloud systems. Therefore, it is important to perform a rigorous study of the existing simulators, in order to select the most suitable ones, to address the problem to be solved. Also, some simulators do not faithfully reproduce the systems under study. That is, the accuracy of the simulation results differs with the selected simulation platform. In order to overcome these disadvantages, we have designed a methodology, presented in Section 3, which aids to select the most suitable simulator depending on the aspect of the cloud under study. Moreover, we have performed a careful analysis of the current literature [4, 25, 63], as a result, six well-known cloud simulators that are widely used by the research community, have been selected and analysed (See Table 2.1). The first column of the table provides the name of the analysed simulators. DC-Sim [96] focuses on simulating a data-centre hosting an Infrastructure as a Service cloud, GreenCloud [170] provides detailed modelling of communication aspects of the data-centre, SimGrid [41] supports the study of scheduling algorithms for distributed applications in heterogeneus distributed systems, iCanCloud [42, 129] predicts the trade-offs between cost and performance of a given application executed in a specific hardware, CloudSim-Plus [65] focuses on avoiding the low-level details related to cloud-based infrastructures and services, CloudSim-Storage [135] provides storage support.

The next three columns, namely *Language*, *License* and *GUI*, refer to the programming language used to develop the simulator, the software license

Simulator	Language	License	GUI	Net	Energy	Sto.	Cost	SLA
DCSim	Java	GPL3	No	Lim.	No	No	Yes	Yes
GreenCloud	C++, OTcl	GPL	Yes	Full	Yes	No	No	Yes
SimGrid	C/C++	GPL	No	Lim.	Yes	Yes	No	No
iCanCloud	C++	GPL3	Yes	Full	Yes	Yes	Yes	No
CloudSim-Plus	Java	Apache 2	No	Lim.	Yes	No	Yes	No
CloudSim-Sto.	Java	Apache 2	No	Lim.	I/O	Yes	No	No

Table 2.1: Comparative study of current cloud simulators

of the simulator and to the support for a graphical interface, respectively. In general, Java and C++ are the most adopted languages to write cloud simulators. Regarding the license, the analysed simulators provide a *free software* license. Next, the functionality of each simulator is analysed.

The column labelled as *Net* determines the capacity of the simulator to model the communication network. In this case, GreenCloud and iCanCloud completely model the communication network, implementing the TCP/IP protocol. Other simulators provide a limited model to simulate the network, which is *acceptable* for simulating cloud systems. In general, a cloud consists of a high number of hosts and communication devices, like switches and routers, leading to long execution times to accurately represent the behaviour of the network. Next, the column labelled as *Energy* shows if the simulator is capable of representing the energetic consumption of the cloud. In this case, DCSim does not support this feature, while CloudSim-Storage only models the energy consumption for the storage devices. The iCanCloud simulator models the energy consumption using the *E-mc*^{2} framework [42]. The capability of each simulator to model the storage system is depicted in the next column. The iCanCloud simulator provides a wide variety of configurations for modelling the storage system, like parallel and distributed file systems and RAID. The CloudSim-Storage simulator is an extension for the CloudSim that focuses on modelling the behaviour of the storage devices, including its energetic consumption. SimGrid also provides models to represent the storage devices in a cloud system. Finally, the last two columns represent, respectively, the capability to model cost and Service-Level Agreements (SLAs). In this case, DCSim is the only simulator that provides support for these features.

As it is shown in Table 2.1, there are no simulators capable of fully modelling a cloud system. In general, each simulation tool focuses in one or several features and, consequently, different simulators must be appropriately combined for investigating the different aspects of the underlying cloud infrastructure under study. For more information about these simulators see Section 3.1.3.1

2.3 Evolutionary algorithms

Evolutionary Algorithms (EAs) are stochastic search algorithms inspired by the process of neo-Darwinian evolution [92], that is, natural selection and natural genetics. Consequently, the use of heuristics and the reliance on stochastic processes are two of the main features of EAs. The former refers to the fact that there is no guarantee for obtaining the global optimum for the optimisation problem, while the latter means that the obtained results are not deterministic and, therefore, different executions using the same input parameters may produce different outputs.

In general, EAs work with a population of individuals, each one representing a potential solution to solve an optimisation problem. The idea is to simulate the evolution of these individuals, so that those more adapted to the environment propagate their *genetic information* to the next generation of individuals. The quality of each individual is evaluated using a fitness function. Thus, the best individuals of each generation have a higher probability of being selected for reproduction. In such a case, the individuals are combined and modified using operations inspired by natural genetics. It is therefore expected that each new iteration (also called *generation*) produces enhanced individuals through natural selection. In general, EAs apply two operations to combine individuals: *crossover* and *mutation*. The former swaps genetic information from two individuals, while the latter randomly modifies the genetic information of one individual.

EAs are based on a generic search paradigm that can be used to solve a wide spectrum of problems. For example, EAs have proven to be robust to deal with non-functional properties, like energy consumption [3]. In this thesis, we use EAs to efficiently guide the search for optimising the energetic consumption of cloud systems. On the contrary, other conventional methods based on local or non-adaptive search, like random or greedy algorithms [169], are unreliable to find feasible solutions for this problem.

The current literature reports different bio-inspired techniques, such as Genetic Algorithms (in short, GAs) [122], Genetic Programming (in short, GP) [13], Ant Colony Optimisation (in short, ACO) [59], Cat Swarm Optimisation (in short, CSO) [11], Particle Swarm Optimisation (in short, PSO) [123] and Bee Colony Optimisation (in short, BCO) [106], among others. Despite the diversity, all these variants are based on a common working scheme, where the performance and accuracy obtained strongly depend on the type and the complexity of the problem [61, 94]. Thus, the task of selecting a bio-inspired technique to solve a computational problem is vital to successfully design a valid solution.

In our proposal, we need to model complex systems that require a high number of inter-related parameters and, therefore, we need flexibility, to generate a wide spectrum of cloud configurations (individuals), and high perfor-

mance, to evaluate them efficiently. Techniques inspired by ACO, CSO, PSO and BCO are based on swarm intelligence, where the focus is on cooperation. In essence, swarm intelligence focuses on those systems containing many individuals that coordinate using decentralised control and self-organisation. Since our individuals must compete to obtain the best result (energy consumption), we discarded these approaches. On the contrary, GAs promote competition, where the individuals more adapted to the environment propagate their *genetic information* to the next generation of individuals. Thus, the best individuals of each generation have a higher probability of being selected for reproduction. GAs have been used in the past to solve problems related to the cloud [178, 181]. Moreover, we think that mutation and crossover techniques are suitable to face the problem of optimising energy consumption in cloud systems. For this, we propose a hybrid encoding – combining GAs and GPs – based on graphs and integer representation that eases the processing of the large structures that conform the cloud.

In summary, the key to successfully solve a problem using EAs is to provide a proper representation of individuals (the potential solutions), a suitable fitness function to measure their quality, and designing effective crossover and mutation operators to guide the search for optimal solutions.

Algorithm 1 shows the generic scheme of an EA, which follows the approach just described. The algorithm proceeds in a loop, which typically stops whenever some individual reaches a certain quality.

Algorithm 1: Generic scheme of EAs.

```
Initialise population of individuals
Measure the quality of each individual using a fitness function
while stop criterion is not reached do
    Select a subset of individuals from the current population
    Apply genetic operators (mutation and crossover) to selected
    individuals
    Calculate the quality of each new generated individual using a
    fitness function
    Update current population
```

2.4 Model Driven Engineering basics

Since one of the main goals of this thesis is to provide a formal model of the cloud using MDE, this section presents some of the fundamental concepts of this paradigm, like a model, a meta-model and the conformance relation.

As we introduced on Section 2.4, MDE is a software engineering paradigm relying on the intensive creation, manipulation and (re)use of several and diverse types of models [184]. A model is *"a system that helps to define and to give answers to the system under study without the need to consider it*

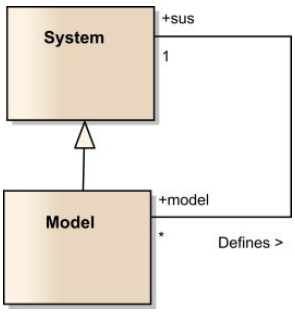


Figure 2.1: Model definition (from [9]).

directly"[9] (See Figure 2.1). That is, a model is a reduced representation of a system that contains the most relevant properties of the system under study. These models are sometimes defined using general-purpose modelling languages, but for restricted, well-known domains, it is also frequent the use of DSLs tailored to the application domain and objectives of the project [83]. One way to define DSLs in MDE is by specifying meta-models, like the one shown in Figure 2.3, which are models that describe the concepts and define the abstract syntax of a DSL (See Figure 2.2). Specifically, a DSL is defined as *"a set of all possible models that are conformant with the modelling languages abstract syntax, represented by one or more concrete syntaxes and that satisfy a given semantics"* [9]. Meta-models are described using notations similar to class diagrams, like the MOF. A model conforms to a meta-model – through the conformity relation – when it satisfies the rules defined at the level of its meta-model [139].

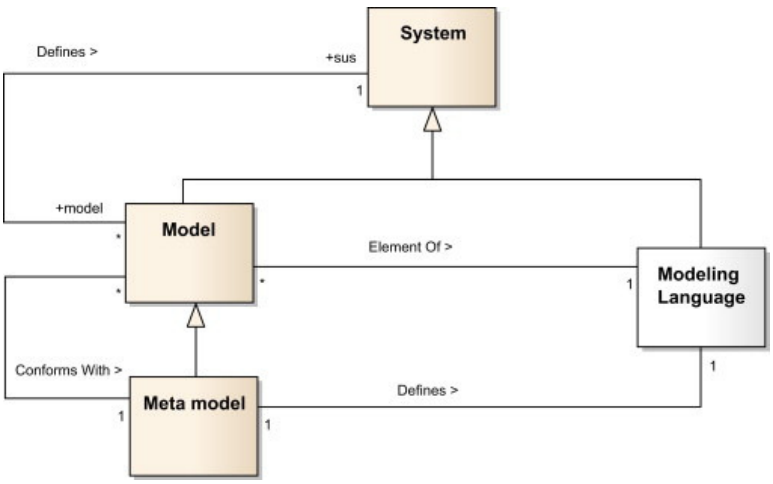


Figure 2.2: Meta-model definition (from [9]).

The notation used to describe the meta-model may not be able to capture all properties of interest for a DSL. For this purpose, it is common to use the

```
1 context Network
2 inv CheckNetworkBandwidth:
3 self.Bandwidth >= 0 and self.Bandwidth <= 32768
```

Listing 1: Invariant for checking the validity of the network bandwidth

OMG's Object Constraint Language (in short, OCL) [150, 177]. The OCL is a formal language that can be used to describe expressions on UML models, or integrity constraints on meta-models. The latter are class invariants that must be fulfilled by the instance models of the meta-models.

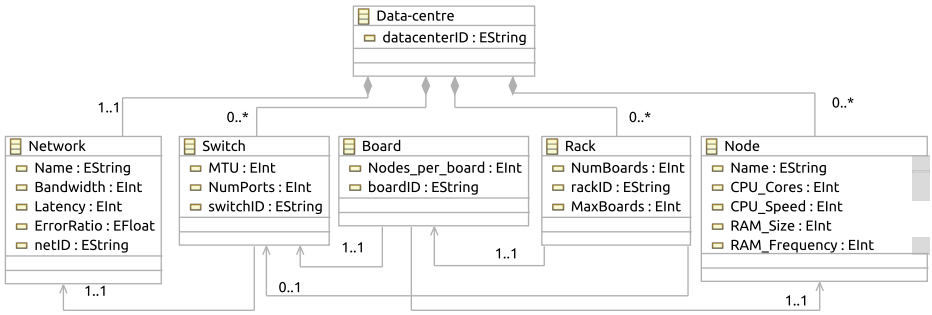


Figure 2.3: Meta-model excerpt for a cloud data-centre.

As an example, Listing 1 describes an OCL invariant for the cloud system meta-model to ensure that the Network Bandwidth of a data-centre is within a valid range of values. Moreover, the context class is Network, because the invariant must check the bandwidth property of its instances.

Listing 2 describes another example of OCL invariant, whose objective is to validate that the attribute *id* is different for objects of type Rack. For this, *forAll* is used to compare the *id* of each two objects of type *Rack*. The *forAll* operation returns *true* if the given condition is satisfied by all instances of the collection.

```
1 context DataCenter
2 inv checkRackNames:
3 self.racks->forAll(r1, r2 | r1<>r2 implies r1.rackID<>r2.rackID)
```

Listing 2: Invariant for checking the validity of the rack identifiers

Chapter 3

A methodology for validating cloud systems using Metamorphic Testing and Simulation

Reach for the stars,
so if you fall,
you land on a cloud.

Kanye West

This chapter presents the contributions in the field of MeT focused on checking the correctness of cloud systems. Section 3.1 presents a study of MeT-based approaches existing in the literature. Section 3.2 describes a methodology for analysing energy-aware properties of cloud systems using EAs. Section 3.3 describes a methodology for the automatic analysis of memory systems, which are considered a key component of the physical machines that support the cloud systems. Section 3.4 summarises the chapter and provides some conclusions.

3.1 State of the art

This section presents a study of the different techniques existing in the literature, which are combined in this thesis with MeT, to analyse the correctness of cloud systems.

3.1.1 Metamorphic Testing for cloud systems

MeT is a testing technique introduced by Chen et al in 1998 [48]. Since its introduction, several techniques, assessment studies and applications have been presented.

During the last years, MeT has been successfully applied as an effective approach to alleviate the oracle problem [49, 161]. Remarkably, MeT was able to detect new faults [146, 183] in three out of seven programs in the Siemens suite [81], which has been studied in major software testing research projects for 20 years. Similarly, Le et al. discovered over one hundred faults in two popular C compilers (GCC and LLVM) using MeT [109].

With respect to the application domain, the first one was performed in the field of numerical programs in 2002. Whilst in the following years the potential applications of MeT were mainly explored in a theoretical way, its application to multiple domains occurred from 2007 onwards. Recently, MeT has been applied in different application domains, including the validation of complex systems [161]. The complexity of these systems hampers its validation using traditional testing techniques, specifically for analysing non functional properties. In order to alleviate this issue, MeT has been successfully applied to several fields such as web services [162], embedded systems [91] and simulation-based systems, among others.

3.1.1.1 Web services

Regarding the field of web services, Sun et al. presented an MeT framework for testing Web services [26]. An experimental phase was conducted to show the feasibility of the framework by testing a representative Web service. Afterwards, the authors presented a methodology based on MeT and software oriented architecture (in short, SOA) that integrates the previous framework to accurately test Web services [27]. This proposal takes advantage of MRs to generate test cases and validate test results automatically. For the experimental evaluation, the authors analysed three well-known Web services, using MuT to inject faults in the systems under study. The results show the suitability of the proposal, which reaches a range between 77.5% and 94.1% of mutants killed.

Quan et al. proposed a methodology to alleviate the oracle problem in testing online search services [186]. For this, the authors present a method based on the logical consistency of several responses instead of analysing each individual response. The search engines must provide similar results in response to similar queries, which can be seen as particular cases of MRs. The methodology automatically checks both quality assessment and quality improvement of search services. The empirical evaluation was conducted by analysing well-known search engines and showed that the accuracy of the search engines varies over time.

Sun et al. developed an MeT tool for web services called MT4WS [28]. This tool automatically checks the correctness of web services using MeT techniques. Furthermore, the authors present an XML-based MR description language, called MRDL, to aid in the formalisation of MRs, the generation of (follow-up) test cases, and the validation of the results. In order to evaluate the validity of this tool, an experimental phase was conducted, revealing that it is effective for testing web services.

Segura et al. presented an MeT approach for the automated detection of faults in RESTful Web APIs [162]. Specifically, the authors designed six patterns that represent the behaviour of typical MRs found in Web APIs. In addition, they describe a methodology for the identification of MRs based on these patterns. For the empirical evaluation of the approach, several academic and commercial Web APIs were analysed. As a result, up to 90 MRs were identified, which were used to generate both random and manual test cases. The results show that the generated test cases were competent at detecting both synthetic and real bugs.

3.1.1.2 Embedded systems

In the field of embedded systems, Kwong et al. presented a proposal for testing middleware-based software [45]. In this approach, random testing and MeT are combined to automatically generate source and follow-up test cases that satisfy the designed MRs. Also, the authors introduce the concept of checkpoints, which are applied for testing the program using the MRs in different phases of the execution. This technique was validated in an experimental setting that consists of a context-sensitive middleware system and an RFID-based location application.

Chan and collaborators used MeT to check both the functional behaviour and the energy consumption of wireless sensor networks [44]. The authors of this work present two MRs for detecting failures, which are based on data consolidation and equivalent consumption of sensor nodes that are close in proximity. Although the MRs focusing on sensor networks must be constantly dealing with energy-saving issues, these cannot be applied to cloud computing systems (the systems we target in this thesis). First, sensor networks are provided with limited batteries that considerably restrict the operations for computation and transferring information. On the contrary, the cloud uses computing and storage nodes that are provided with a constant power supply. Second, the cloud focuses on virtualising the hardware of computing nodes, allowing several users to share the resources of the same machine, which cannot be applied to sensor networks due to computing power limitations. Finally, the cloud is deployed using a predefined network topology, while the topology of the sensor networks may be built at run-time, allowing variations when the nodes have low battery.

Fei-Ching et al. presented a method for testing wireless embedded soft-

ware. For this, the authors use MeT for validating the correctness of the meter reading function of the wireless metering system [62]. The experimental phase revealed that the proposed MR was effective in detecting, diagnosing and debugging real-life system failures. This fact shows the effectiveness of MeT for improving the quality of the wireless metering systems. Jiang et al. proposed several MRs to ensure the correctness of CPU schedulers [91]. The experimental phase was performed using two simulators, where several failures were injected using MuT. As a result, two faults were detected in one of the simulators under study. This proposal is relevant and it is related to this doctoral thesis, since it analyses one of the important subsystems of the physical machines that supports the cloud systems, the CPU system. In order to complement it, and therefore, analyse another of the important subsystems, one of the targets of this thesis is to provide a set of MRs to analyse functional and non functional properties of memory systems. It should be noted that these MRs are general and are focused on analysing the correctness of the memory of any system, such as cloud systems.

3.1.1.3 Simulation-based systems

Núñez and Hierons [128] combined the iCanCloud simulator with MeT to detect unexpected behaviours when simulating cloud provisioning and usage. Although this contribution provides interesting ideas for checking the correctness of cloud systems, it also presents some limitations. This proposal uses MeT for analysing energy-aware cloud systems, but it only proposes one MR. Since cloud systems are complex, it is necessary to design a complete set of properties to properly check them. That is, this set must consist of diverse MRs, each one targeting a different subsystem of the cloud. Moreover, a reduced number of test cases were applied during the testing process.

Murphy and collaborators presented an approach to systematically test simulation software, specifically for the domain of health care, with the aim of discovering defects in the implementation [124]. Ding and collaborators investigated the effectiveness of MeT to test a Monte Carlo modelling program for heterogeneous media [57]. Additionally, they evaluate the adequacy of testing coverage criteria to measure the quality of the MeT process, to guide the creation of MRs, for generating test inputs and investigate the exceptions found. Chen and collaborators propose the application of MeT to check the conformance between network protocols and network simulators [50], and applied MeT to discover faults in open queuing network models [51]. Altogether, even though these works combine MeT with simulation techniques, MeT has not been appropriately applied to check the correctness of energy consumption in cloud systems.

3.1.2 Evolutionary Algorithms in cloud design and operation

EAs have been successfully applied to solve real-world problems in a wide spectrum of fields, like problem solving from nature, swarm robotics, hardware design and fault tolerance and reconfigurability [33, 185]. In particular, several works applying EAs to cloud systems can be found in the literature as well. Keshanchi and collaborators proposed an improved genetic algorithm for static task scheduling in cloud environments [98], with the purpose of assigning subtasks to processors. A profile-based approach was developed by Vasudevan and collaborators for energy-efficient application assignment to VMs with consideration of resource utilisation [175]. The approach is based on a Repairing Genetic Algorithm (RGA) to solve a large-scale optimisation problem. Xiao and collaborators [182] proposed a novel algorithm based on evolutionary game theory that successfully addresses the challenges faced by the dynamic placement of VMs. In this work, the authors demonstrate that the energetic consumption of a cloud is reduced by dynamically adjusting the placement of VMs. A dynamic task scheduling algorithm that uses an Integer Linear Programming (ILP) model for minimising the energy consumption in a cloud data-centre was developed by Ibrahim and collaborators [84]. The authors of that work also propose an Adaptive Genetic Algorithm (AGA) to reflect the dynamic nature of the cloud environment, which provides a near-optimal scheduling solution that minimises energy consumption.

Drummond presented a GPU-based metaheuristic for workflow scheduling on clouds [60]. The proposal takes advantage of the massive parallelism achieved with GPUs, which allows reducing the computational time of the scheduling process in two orders of magnitude. Chen et al. proposed a predictive and evolutionary approach for cost-effective and deadline-constrained workflow scheduling over distributed cloud infrastructures [47]. For this, approaches based on prediction of time-series are used to feed an EA with performance predictions, to capture dynamic performance fluctuations, and generate schedules in real-time. Ismayilov and Topcuoglu presented a Multi-Objective EA based on neural networks for dynamic workflow scheduling in cloud computing [85]. This work relies in exploiting the history of Pareto-optimal sets to perform predictions after a change. In addition, five existing dynamic algorithms are adapted for the dynamic workflow scheduling problem. Jatoh et al. presented an optimal fitness aware cloud service composition using a genetic algorithm based on adaptive genotypes [87]. Since the system requires several QoS parameters, the solutions need to balance them, and satisfy the connectivity constraints of the service composition.

These works use EAs to optimise cloud systems but focus on managing and scheduling tasks. The approach in this thesis uses MRs – previously designed by an expert – to adapt the search of an optimised cloud configuration using (from the energy consumption point of view) an EA. To the best of our knowledge, there are only a few works in the literature combin-

ing EAs and MeT. Segura and collaborators presented a proof of concept to automate the detection of performance bugs by combining MeT and search-based techniques [163]. Rounds and Kanewala identified 17 MRs for testing a GA and show, through MeT, that these relations are more effective at finding defects than traditional unit tests based on known outputs [153]. Arora and Bassi [10] show that using GAs increases the efficiency of MeT to detect faults in software. However, these works apply MeT to check and test evolutionary algorithms, while our approach focuses on the evaluation of the energy consumption of cloud systems.

3.1.3 Simulation

This section presents a study of the the existing proposals in two simulation fields, cloud systems and memory systems.

3.1.3.1 Simulation of cloud systems

The research community has developed a vast collection of tools for modelling and simulating cloud systems. However, only a small subset of them targets the analysis of the energy consumption of the cloud [25]. This set includes, among others, CloudSim [29], DCSim [96], GreenCloud [170], SimGrid [41], iCanCloud [42, 129] and DISSECT-CF [95].

CloudSim is an extensible and open-source Java simulator, which enables modelling cloud computing systems and application provisioning environments. CloudSim is considered the *de facto* standard cloud simulation platform due to its capabilities for simulating cloud systems, such as VM allocation and provisioning, energy consumption, federated clouds and the possibility to model different types of clouds like public, private, hybrid and multi-cloud environments. One of the key features of CloudSim is the possibility to include new functionalities using extensions like *cloudSim-Storage*, which supports modelling the energy consumption of the storage system [135].

DCSim, also known as *The Data Centre Simulator*, is a Java extensible simulation framework for simulating a data-centre hosting. In essence, DCSim focuses on the IaaS layer for providing services to multiple tenants. GreenCloud is an open-source tool for simulating data centres with a special emphasis on data communication and energy cost in cloud computing. GreenCloud provides a wide range of network and communication configurations for simulating data centres. SimGrid is a tool for simulating algorithms and distributed applications in distributed computing platforms. The resources are modelled by their latency and service rate, and the topology is configurable by the users. Initially, SimGrid targeted grid environments. However, its current version supports a variety of cloud computing use cases including multi-purpose network representation, VM abstraction, live migra-

tion, VM support and storage.

iCanCloud is a simulation platform aimed to model and simulate cloud computing systems by providing different functionalities like resource provisioning. Additionally, the framework *E-mc*^{2} [42] can be used for analysing energy consumption. The main goal of iCanCloud is to predict the trade-offs between cost and performance of a given set of applications executed in specific hardware. DISSECT-CF is a simulator directed to evaluate the energy consumption of IaaSs. DISSECT-CF offers two major benefits: a unified resource-sharing model, and a complete IaaS stack simulation, which includes VM image repositories, storage and in-data-centre networking.

In general, the current approaches for modelling and simulating cloud systems are suitable to represent the behaviour of cloud architectures. However, each simulation tool targets on a specific part of the cloud (e.g. storage system, VMs allocation policies, energy consumption) and, unfortunately, there is no common solution that satisfies the entire research community. Moreover, these tools lack a formal approach to represent cloud systems, which makes difficult to automate the testing process. Our proposed approach focuses on alleviating these issues. First, using different simulation tools allows increasing the features of the cloud infrastructure that can be modelled and simulated. Second, combining MeT and simulation allows to formally model the main features of the cloud and, therefore, automating the testing process. Finally, our methodology can propose cloud optimisations from the energy point of view, which is not supported by these tools.

3.1.3.2 Simulation of memory systems

During the last decade, simulation tools have gained popularity to model and analyse memory systems. Rosenfeld et al. presented DRAMSim2, whose main strength is its accuracy for simulating DDR2/DDR3 models [151]. Also, DRAMSim2 provides different models to represent the energy consumption of the simulated memories. The main weakness of DRAMSim2 is the lack of mechanisms to deploy different memory management policies.

The Utah SImulated Memory Module (in short, USIMM [46]) is a trace-based memory system simulator for on DDRx memories. USIMM provides mechanisms for modelling the different components of the memory system, such as the system architecture, DRAM timing and latency parameters, scheduling policies and power consumption. Moreover, USIMM includes some of the most used PARSEC benchmarks [19].

Jeong et al. proposed DrSim, a simulation platform for modelling DRAM systems, which provides a widespread spectrum of memory architectures and topologies [88].

Similarly, Kim et al. [99] presented Ramulator, a fast and cycle-accurate DRAM simulator that supports an extensive spectrum of DRAM standards,

such as DDR3/4, LPDDR3/4, GDDR5, WIO1/2 and HBM. The main advantage of this simulator is its performance, which appoints Ramulator as the fastest memory simulator. However, several weaknesses like the high abstraction level of the memory components and the lack of both power consumption models and memory management policies, make Ramulator not appropriate for our proposed system. Although these simulators support modelling and simulation of memory systems, the testing process must be manually performed.

3.2 A methodology for analysing energy-aware properties using Metamorphic Testing techniques and Evolutionary Algorithms

This section presents a novel methodology for analysing and improving energy-aware cloud systems using MeT and EAs. First, the main steps of the methodology are described in detail. Next, we present the tool support for the methodology, illustrating the integration of its main components. Finally, we introduce the experimental phase and the validation of the methodology.

3.2.1 Methodology

This section describes a methodology that combines MeT, simulation and EAs to check the correctness of energy-aware cloud systems. The main goal of the methodology is two-fold. First, deciding the most appropriate simulator to model and simulate cloud infrastructures. The high number of existent cloud simulators makes it difficult to select an appropriate simulator for a specific purpose. Thus, this thesis uses MeT to alleviate this issue. We accurately model the cloud in the form of MRs, which represent the underlying behaviour of the cloud. The proposed methodology allows measuring the adequacy of each simulator for simulating cloud systems. A second objective of our methodology is the automated optimisation of the energy consumption of cloud infrastructures. The intention is to apply an EA for evolving cloud systems efficiently, guiding the search using MRs for finding an optimised cloud. Therefore, each new generation of cloud individuals is created by following the constraints defined in the MRs. The main steps of the methodology are depicted in Figure 3.1.

Initially, the features having a relevant impact on the energy consumption must be carefully analysed, like the computing system, the storage system, network features and the workload to be processed, among others. Next, these features are used to design the MRs (label ① in the Figure). The idea is to provide a formal and accurate model – in the form of MRs – that represents the underlying behaviour of the cloud. The set containing the

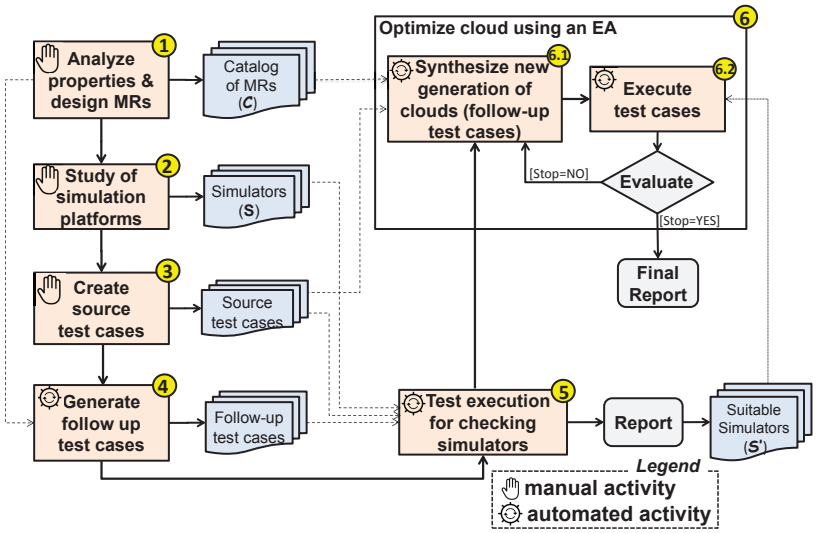


Figure 3.1: Scheme of the methodology.

provided MRs, which we refer to as *catalog*, is denoted by $\mathcal{C}$.

This methodology is not based on a specific tool, and therefore, it is desirable to use several simulators in the testing process. To that end, the tester must choose the simulators that offer capabilities to model and simulate the features formulated in $\mathcal{C}$ (label ②). In this step, the simulators are not executed, but their specifications are carefully analysed to determine whether they can be used in the testing process. The set of simulators chosen by the tester in this step is denoted by $\mathcal{S}$.

As it was previously introduced in Section 1.1.1, an MR is a formula $\mathsf{i}(MR) \implies \mathsf{o}(MR)$, where $\mathsf{i}(MR)$ is a relation between the source test case and a follow-up test case, and $\mathsf{o}(MR)$ is a relation over the results obtained from the execution of these test cases. The input relation $\mathsf{i}(MR)$ must be fulfilled by both the source test case and the follow-up test case.

In essence, a test case consists of a cloud model and a workload. The cloud model contains details about the underlying architecture of the system, such as the number of physical machines, the features of the communication network, features of each physical machine (i.e. CPU, storage, memory, etc.), configuration of VMs and resource allocation policies, among others. The workload represents the operations performed by the cloud, that is, requests of VMs to be deployed on physical machines, storage operations and computing operations. A test case is a tuple (m, ω), where m refers to a cloud model and ω refers to a workload that is executed over m. Similarly, a follow-up test case is denoted by (m', ω').

The execution of a workload ω over a cloud model m is carried out by simulation. Thus, we denote by $S(m, \omega)$ the result of the simulation – using the simulator S – for executing the workload ω over the cloud m.

A test case is formally defined as $t = (m, \omega)$ and we write $(t, f, S) \models MR$ to denote that a follow-up test case $f = (m', \omega')$ satisfies an MR in a simulation performed by simulator S. Intuitively it means that whenever the source test case t and the follow-up test case f satisfy the conditions in $\mathsf{i}(MR)$, then the outputs obtained from $S(m, \omega)$ and $S(m', \omega')$ satisfy the conditions in $\mathsf{o}(MR)$.

Next, in step ③, the tester manually designs a reduced number of source test cases. This set is called $\mathcal{T}$. Basically, the main difference between a source test case and a follow-up test case lies in the way the test case is generated. While a source test case is manually designed by the tester, a follow-up test case is automatically generated by using a source test case and an MR. In step ④, we apply a procedure to generate a set $\mathcal{F}$ of follow-up test cases.

The main goal of the next step (label ⑤) is two-fold: first, to analyse the adequacy of each MR; second, to investigate how appropriate is each simulator to represent the energy consumption of cloud systems. In order to accomplish these objectives, the source test cases and the follow-up test cases are executed in the simulators chosen in step ②. We measure the adequacy of an $MR \in \mathcal{C}$ by calculating the percentage of follow-up test cases f, generated from each source test case $t \in \mathcal{T}$ and executed using the simulator S, that fulfill $(t, f, S) \models MR$. The adequacy of a metamorphic relation MR using a simulator S and the test selection strategy $\mathcal{T}$, written $adq_{\mathcal{T}}(MR, S)$, is a number between 0 and 1 calculated as follows:

$$adq_{\mathcal{T}}(MR, S) = \frac{\sum_{t \in \mathcal{T}} |\{(t, f, S) \mid f \in followUp(t) \ \wedge \ (t, f, S) \models MR\}|}{\sum_{t \in \mathcal{T}} |\{(t, f, S) \mid f \in followUp(t)\}|} \tag{3.1}$$

where $followUp(t)$ is the set of generated follow-up test cases for t.

Once all the tests are executed over the simulators, a report containing the adequacy of the MRs is generated. Next, the tester uses this report to create a new list of simulators, discarding those that do not appropriately represent the behaviour of cloud systems. As a result, a new list of simulators, denoted by $\mathcal{S}'$, is generated.

The quality of the cloud designs – focusing on energy consumption – that were created by the tester in step ③, is optimised in the last step (label ⑥). In order to accomplish this task we use an EA. The EA evolves a population of individuals – cloud models in this case – until one of them fulfils the stop criteria (e.g. the energetic consumption of the resulting cloud has been reduced a 5%).

The initial population (see label 6.1 in Figure 3.1) is generated from the source model, which is provided by the user. Thus, using this cloud as a basis, the EA applies different operations to evolve each individual and create a new generation of clouds. Once a new generation is created, each

cloud is analysed using a fitness function. If a satisfactory solution is reached, the algorithm stops and the resulting cloud is reported to the user. On the contrary, a new generation of clouds is generated and each one of them is evaluated.

3.2.2 Tool support

We have developed tool support for the proposed methodology, which implements the different modules illustrated in Figure 3.1 using Java. Its scheme is depicted in Figure 3.2, and shows how MeT, EAs and simulation tools are combined to optimise the energy consumption in cloud systems. For the sake of clarity, only the most relevant parts are shown in this scheme.

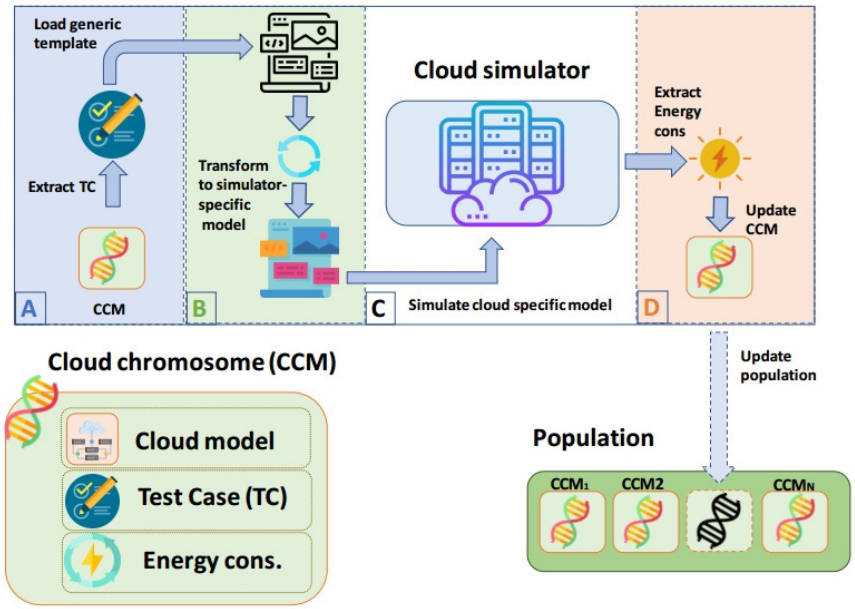


Figure 3.2: Tool support for the methodology: integration of the simulation tools within the EA scheme.

The Evaluation Module (in short, EM) consists of 4 main submodules: test case extraction module (labelled as A), template transformation module (labelled as B), cloud simulation module (labelled as C) and energy consumption module (labelled as D).

Initially, the *cloud chromosome* (in short, CCM) is extracted from the population. A CCM consists of three main elements: a *cloud model* (in short, CM), a *test case* (in short, TC) and *energy consumption* (in short, EC). Test cases are automatically generated using MeT techniques. Each TC contains a CM and a workload, where CM represents all the components used to model a cloud system, such as computational resources, network topology, and the workload refers to the operations to be processed by the CM. Each

object that represents a TC contains a path indicating the location of the test input data and a path where the results of the simulation are stored. The last element of the CMM, labelled as *Energy consumption*, refers to the amount of energy required by the CM to execute the workload.

The information allocated in TC is parsed from the CCM to create a generic data structure. It is *generic* since it allocates the data required to configure a test case, but is not specific to a single simulator. The idea is to manage a common structure that can be applied to the different simulators deployed in the system, allowing an easy translation of this data into the different formats of the cloud simulators that are currently deployed (module A).

Next, the configuration files required to execute the TC using a cloud simulator are generated (module B). The cloud simulator deployed in module C executes the simulation of the CM contained in TC. Once the simulation has finished, a results file is generated (see module D). The energy consumption is extracted from the results, and the CCM is updated and inserted into the population accordingly.

3.2.3 Experiments and validation

In order to validate the methodology proposed in this section, we have used 8 MRs and chosen different well-known simulators designed to model and simulate the energy consumption of cloud systems. First, we have carefully investigated the main features and drawbacks of each simulator, and analysed the documentation provided by each simulator to decide whether a simulator is appropriate, or not, for this study. As a result, we provide the set of 7 simulators: CloudSim, CloudSimStorage, DCSIM, GreenCloud, simGrid, iCanCloud and DISSECT-CF.

Next, we manually design three clouds – *cloudA*, *cloudB* and *cloudC* – providing different configurations, each one representing a specific cloud profile. Thus, *cloudA* represents a low-profile cloud, providing slow CPUs, small RAM memories and a slow network; *cloudB* models a high-profile cloud, with large RAM memories, fast CPUs with 8 cores and a fast communication network; and *cloudC* represents a mid-profile cloud, with a fast communication network and a fair CPU and memory systems. Additionally, we have created different workloads, which are inspired by operations performed in big data analysis. In the following, a trace is denoted by ω_{sim}^{size}, where *sim* is the simulator used to execute the trace and *size* represents the trace length. The size of a small trace is denoted by the sub-index s, the size of a medium trace – larger than the small trace – is denoted by m, and the size of the largest trace is denoted by l.

The set of source test cases is generated by combining the selected clouds and the three generated workloads. In step 4, we automatically generate a

set of follow-up test cases, containing a total of 4000 follow-up test cases.

Regarding the adequacy, all simulators used in the study provide acceptable results to simulate cloud systems. Hence, we chose cloudSimStorage and simGrid as the most appropriate simulators, for two reasons. First, they provide high performance for executing the simulations. Second, the obtained results show that these simulators are suitable to model and simulate the required features of cloud models.

Finally, we conducted an empirical study where three different cloud systems have been analysed using two well-known simulators, simGrid and cloudSim. With respect to the workload, we use traces that represent the infrastructure of PlanetLab [142], to be executed by cloudSim, and a Map-Reduce based application [100], to be executed by simGrid. For the sake of clarity, we summarise the results obtained from this study in Tables 3.1 and 3.2. In essence, we use two different approaches to test the clouds. The first approach (labelled as EA) uses our EA to find a proper optimisation of the cloud under test, while the second approach (labelled as Random) randomly applies a mutation operator to the cloud under test for generating new cloud configurations.

Configuration		EA			Random		
Cloud	Workload	min	max	avg	min	max	avg
cloudA	$\omega^{s}_{cloudSim}$	702.99	1189.99	1001.20	1323.25	1443.94	1331.14
	$\omega^{m}_{cloudSim}$	944.52	1200.77	1114.78	1333.27	1566.17	1353.12
	$\omega^{l}_{cloudSim}$	1094.73	1223.34	1137.89	1354.37	1800.84	1412.05
cloudB	$\omega^{s}_{cloudSim}$	723.09	1184.74	987.65	1318.06	1374.83	1320.46
	$\omega^{m}_{cloudSim}$	933.82	1190.40	1104.28	1322.89	1455.99	1339.29
	$\omega^{l}_{cloudSim}$	691.72	1201.55	1030.47	1332.74	1546.55	1341.36
cloudC	$\omega^{s}_{cloudSim}$	581.43	1316.57	869.43	1449.42	1827.10	1498.49
	$\omega^{m}_{cloudSim}$	1207.75	1455.49	1318.36	1449.42	1598.15	1566.77
	$\omega^{l}_{cloudSim}$	1146.80	1466.18	1303.45	1395.83	2165.27	1817.95

Table 3.1: Pseudo-random versus EA approach using cloudSim

Configuration		EA			Random		
Cloud	Workload	min	max	avg	min	max	avg
cloudA	$\omega^{s}_{simGrid}$	206.99	903.71	600.27	668.37	1065.15	939.68
	$\omega^{m}_{simGrid}$	1815.57	3500.75	3116.08	2825.67	3895.14	3551.99
	$\omega^{l}_{simGrid}$	1761.64	23863.45	16479.02	15223.84	29083.71	23419.71
cloudB	$\omega^{s}_{simGrid}$	196.95	806.33	540.37	489.79	1022.51	943.46
	$\omega^{m}_{simGrid}$	666.44	1882.89	988.29	1350.58	1997.83	1856.99
	$\omega^{l}_{simGrid}$	3174.85	11926.45	9479.64	8671.07	12823.77	12259.79
cloudC	$\omega^{s}_{simGrid}$	348.22	922.13	636.67	549.92	1009.09	845.64
	$\omega^{m}_{simGrid}$	**1763.21**	2724.51	2478.34	**1577.35**	2921.28	2617.96
	$\omega^{l}_{simGrid}$	6034.08	15257.92	12389.22	9118.56	15257.92	12829.67

Table 3.2: Pseudo-random versus EA approach using simGrid

In the tables, the first column represents the cloud under test and the executed workload. The second column depicts the results provided by the EA, while the third column shows the results provided by the random ap-

proach. Each value represents the energy required by the cloud to execute the workload. Thus, *min* refers to the "best" cloud configuration generated, in the sense of energy consumption, using as a basis the cloud under test, *max* refers to the "worst" generated cloud and *avg* refers to the energy consumption average of all the generated cloud configurations – using the cloud under test as basis – to execute a specific workload.

After a careful analysis of the results, we observe that our EA clearly provides better results than the random approach. We can conclude that our methodology is able to provide different alternatives to improve the energy consumption and automatically detect flaws in the cloud designs created by the user.

The complete results and the definition of the MRs used in this study can be consulted in the paper associated with this contribution [30] (7.2).

3.3 A methodology for checking the correctness of memory systems

In this section, we describe a methodology for checking the correctness of memory systems using MeT and simulation techniques. This methodology is described in Section 3.3.1 and the supporting tool is instroduced in Section 3.3.2. The experimental results obtained in the empirical study are described in Section 3.3.3. Finally, some conclusions are provided in Section 3.4.

3.3.1 Methodology

This section presents a description of the proposed methodology for checking the correctness of memory systems, which are considered an important subsystem for the overall performance of a cloud system. The methodology combines simulation and MeT to test memory systems. Figure 3.3 shows its main steps, which are detailed in the following subsections.

In the first place, a memory model needs to be defined (label 1), and as in any testing process, a set of input test cases need to be provided (label 2). In case of memory systems, benchmarks inspired by the PARSEC suite can be used [19]. The methodology assumes the availability of a number of MRs. In the following step (label 3) the tester selects a MR to be used as oracle, and follow-up test cases are generated (label 4) to exercise the chosen MR. In step 5, the memory management algorithm is simulated on the memory model using both the input test cases and the follow-up test cases. The simulation provides outputs, typically informing about the consumed power, time, and number of read/write operations. Then, these results are compared according to the criteria given by the MR (label 6). If they do not match, it means an error has been found. If they match, the confidence on the

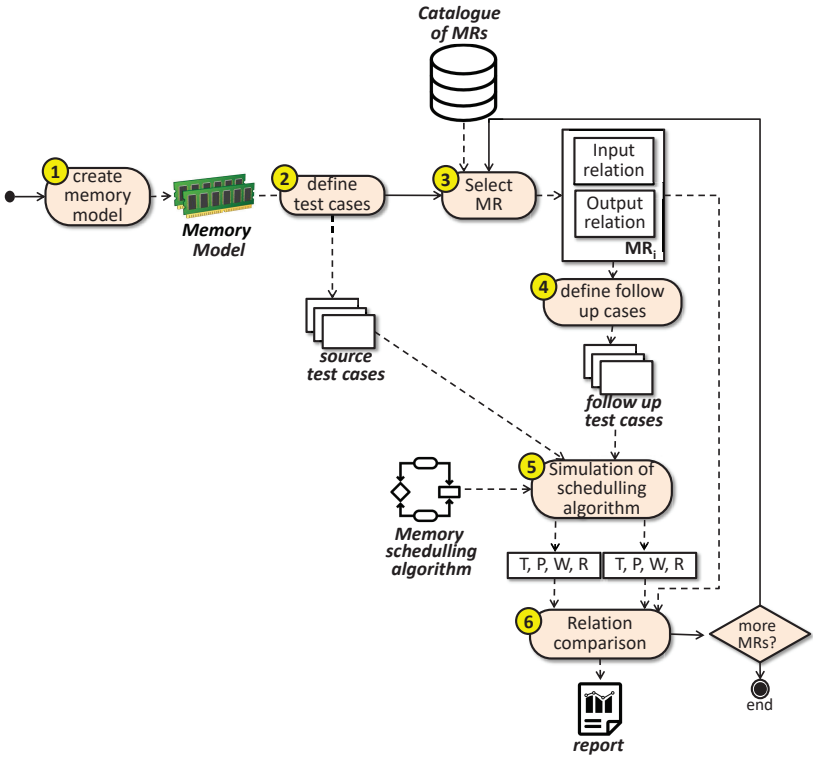


Figure 3.3: Scheme of the methodology.

correctness of the memory system increases, and new MRs can be selected for testing.

3.3.2 Tool support

This section describes the tool that supports the methodology introduced in the previous section. The novelty of the tool lies in an expert system to properly analyse memory systems.

An expert system is a computational system that emulates the decision-making of human expertise using domain-specific knowledge. The main difference between an expert system and a conventional system lies in the method used to solve complex problems, that is, while expert systems apply reasoning based on rules, conventional systems are based on procedural code.

In contrast with the conventional architecture of expert systems, we have included two additional modules: a factual database and a simulation platform. Thus, our proposed system consists of 5 main modules (see Figure 3.4).

The *knowledge base* (in short, *KB*) is a module that is built using the knowledge of the expert. In essence, *KB* consists of rules and facts. In this case, the rules are introduced into the *KB*, by the expert, in the form of

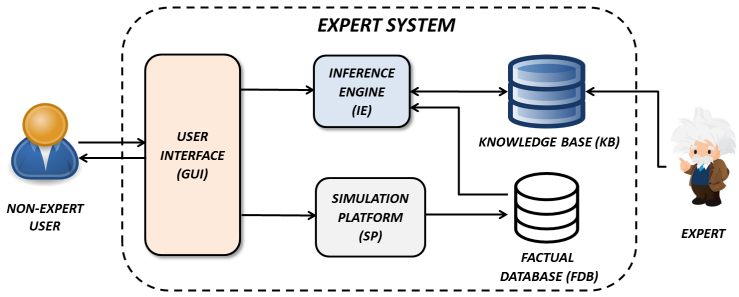


Figure 3.4: Architecture of the proposed expert system.

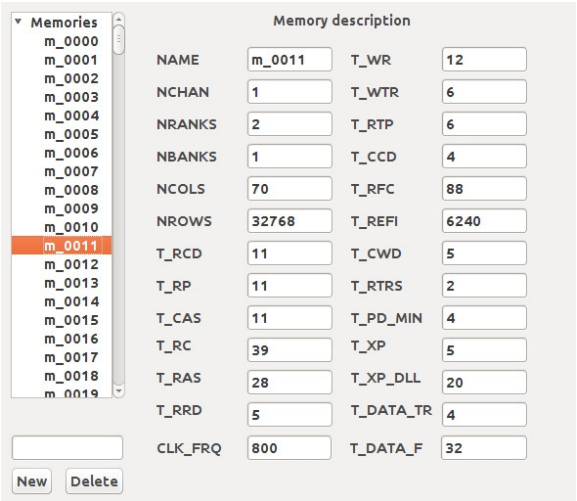


Figure 3.5: Editor for modelling memory configurations.

MRs.

The *user interface* module is a friendly and easy-to-use application - written in Java - that provides a graphical user interface (in short, *GUI*). Using this *GUI*, non-expert users can perform different tasks like modelling new memory systems, editing the configuration of a current memory model and testing memory models. Figure 3.5 shows the editor for modelling memory systems. Basically, this editor contains each parameter of the memory model and its corresponding value. The left part of the panel shows a repository of memory models, where users can easily save, edit and remove memory models in the application.

Figure 3.6 shows a panel that allows users to select the MRs that will be used in the testing process. These rules are obtained from the *KB* and displayed in the *GUI*. Thus, if the expert updates the rules in the *KB*, these are also updated in the *GUI*.

Figure 3.7 shows the results of testing a memory system containing faults. In this case, the expert system detects faults in the read queue, write queue

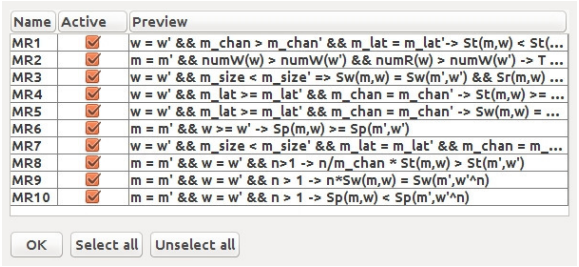


Figure 3.6: Panel to select the MRs used in the testing process.

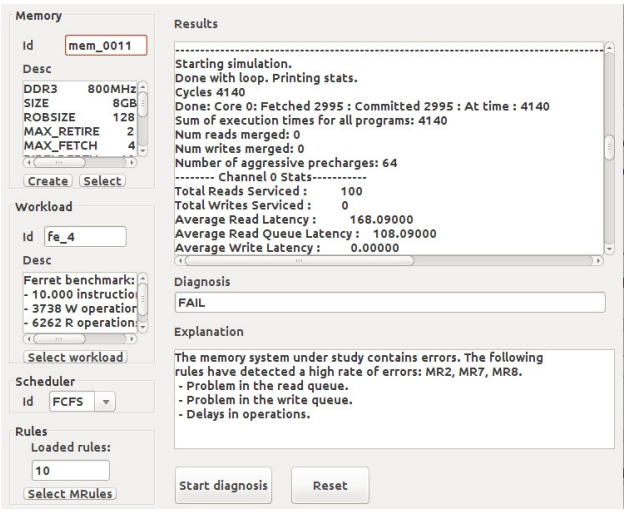


Figure 3.7: Results of a faulty memory system.

and delays in the operations. Similarly, Figure 3.8 shows the results of testing a correct memory system.

The *simulation platform* (in short, *SP*) is in charge of two main tasks. First, once the non-expert user has defined a memory model and selected the required MRs, the *SP* uses this information to automatically generate a set of follow-up test cases. Second, for each generated follow-up test case, the *factual database* (in short, *FDB*) is accessed to request information of the test. If the test is stored in the *FDB*, then the required information is obtained from the *SP*. In other case, the *SP* executes the simulation of the test case to produce the results and to extract the required information to be stored in the *FDB*.

The facts and rules – also called MRs – are analysed by the *inference engine* (in short, *IE*) and, for each test case, the *IE* checks if the involved MRs in the testing process are fulfilled by matching the obtained outputs.

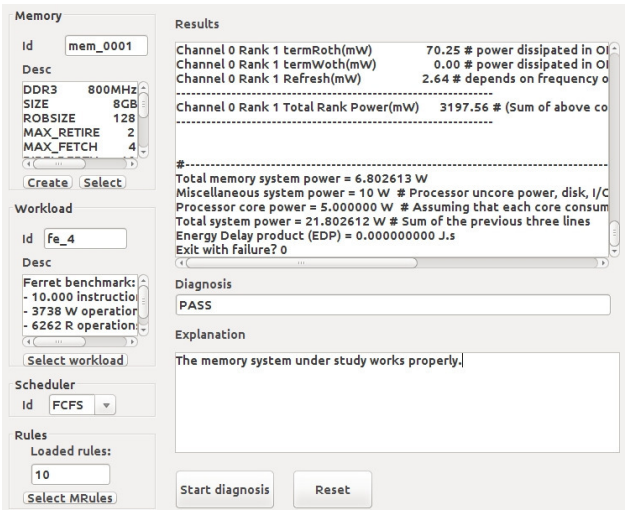


Figure 3.8: Results of a correct memory system.

3.3.3 Experiments and validation

The main goal of the proposed methodology is to check the correctness of memory systems using realistic memory models with a high level of detail. Hence, even though our system is general and independent of a specific simulator, we consider USIMM as the most suitable option for the purposes of our study. First, this simulator provides a good compromise between the high level of detail in the hardware models and the inherent flexibility to model a wide range of memory systems. Second, USIMM supports simulation using customised memory management policies. In fact, USIMM has been used in the Memory Scheduling Championship [46]. Third, this simulator provides an accurate power consumption model and generates a detailed collection of statistics as output.

For the validation of the methodology, two experiments have been designed, each with 500 memory models. Also, 50 different workloads have been created, which are inspired by PARSEC benchmarks, such as *blackscholes, facesim, ferret, fluidanimate, freqmine, streamcluster and swaptions*. In order to generate traces that are representative of the selected benchmarks, the SimPoint platform has been used [166]. During the trace generation process, SimPoint uses basic block vectors to recognise execution intervals that can be used to reflect the behaviour of the benchmark.

Each memory model generated in this study has been tested using 10 MRs and 4 different memory management algorithms. Two of these algorithms are based on well-known scheduling policies: first come, first serve (in short, FCFS) and an approach based on the close page policy (in short, CPP). The other 2 management algorithms won the Memory Scheduling Championship: high performance memory access (in short, HPMA) and re-

quest density aware fair memory (in short, RDAF) [12].

We are particularly interested in investigating the effectiveness of random testing to detect faults in memory systems. Hence, we have carried out an experiment where 1000 test cases are randomly generated for testing the four different memory scheduling algorithms analysed in previous sections (CPP, FCFS, HPMA and RDAF). Again, for each algorithm, the same five faults were artificially injected. Similarly, other 1000 test cases have been generated, using our proposed system, to detect the same faults. The idea is to compare the effectiveness to detect the injected faults of our system against the one reached by random testing.

Table 3.3 shows the results of this study, where each column - namely *Fault$_1$* to *Fault$_5$* - represents the different injected faults in the memory schedulers. Each row represents the system under test (in short, SUT) and consists of two different values, where *valid* shows the percentage of test cases that are successfully executed and *detect* is the percentage of *valid* test cases that detects the fault. For instance, in order to test the CPP scheduler, random testing generates 31 valid test cases – from the 1000 test cases generated in total – where only 58.06% of these test cases (18 in total) are able to detect Fault$_1$.

SUT		Fault$_1$		Fault$_2$		Fault$_3$		Fault$_4$		Fault$_5$	
		Rnd	ES	Rnd	ES	Rnd	ES	Rnd	ES	Rnd	ES
CPP	Valid	3.10	99.20	2.90	99.60	3.00	99.20	3.10	99.01	3.20	99.41
	Detect	58.06	100	89.65	100	90	100	48.38	100	41.93	100
FCFS	Valid	3.10	99.10	2.90	99.80	2.90	99.65	3.00	99.90	3.10	99.10
	Detect	90.32	100	100	100	89.65	100	60.00	100	100	100
HPMA	Valid	3.00	99.6	2.90	99.5	3.10	99.40	3.10	99.5	3.10	99.6
	Detect	60.00	100	27.58	100	58.06	100	60.00	100	19.35	100
RDAF	Valid	3.00	99.6	2.90	99.5	3.10	99.4	3.10	99.5	3.10	99.6
	Detect	20.00	99.8	37.93	100	90.32	100	67.74	100	27.58	100

Table 3.3: Effectiveness (in %) of random testing vs. our proposed system using composed MRs

Overall, random testing provided a lower effectiveness to test the memories than our proposed system and requires a considerable effort to check the provided results. Hence, we think that our proposed methodology is a valuable contribution for the research community, not only for automatically generating quality test cases, but to alleviate the oracle problem, eliminating the effort of the tester to check the provided output.

The complete results of the study can be consulted in the paper associated with this contribution [38] (7.1).

3.4 Summary and conclusions

This chapter has presented the techniques combined with MeT to accomplish the first objective of this thesis: providing methodologies to analyse the correctness of cloud systems. This objective can be divided in two parts,

each one providing a specific methodology.

The first one is a methodology that combines MeT, simulation and EAs to check the correctness of energy-aware cloud systems. The second one is a methodology for analysing the correctness of memory systems combining MeT and expert systems. The knowledge of the expert is represented in the form of MRs, which are properties of the analysed system involving multiple inputs and their outputs. It should be noted that both methodologies have common aspects, such as the MRs, the generation of follow up test cases and the comparison engine.

The proposed methodologies have been designed to be flexible and scalable. In this way, several MRs, algorithms to generate follow-up test cases and simulation platforms, can be easily included. These methodologies can be used to analyse – at runtime – functional and non-functional properties of the cloud models designed along with this thesis.

3.5 Associated publications

(7.1) **An expert system for checking the correctness of memory systems using simulation and metamorphic testing**
Pablo C. Cañizares, Alberto Núñez and Juan de Lara.
Expert Systems with Applications 132: pp. 44-62 (2019)

(7.2) **MT-EA4Cloud: A Methodology for testing and optimizing energy-aware cloud systems.**
Pablo C. Cañizares, Alberto Núñez, Juan de Lara and Luis Llana.
Journal of Systems and Software (under 3^{rd} round of review)

Chapter 4

Mutation Testing techniques for analysing cloud systems

Next time you feel
like your world's about to end,
I hope you studied
because he's testing your faith again.

Kendrick Lamar

This chapter presents the contributions made in the field of MuT. Initially, Section 4.1 presents a study of different MuT techniques applied to distributed systems and performance. Next, Section 4.2 presents the basic architecture of MuTomVo, a MuT framework for simulated cloud and distributed systems. Section 4.3 describes a set of optimisation strategies that improves the overall performance of the MuT process. Finally, Section 4.4 summarises the proposals and finishes with the conclusions.

4.1 State of the art

In this section, we first review current approaches applying MuT to distributed systems, and then we analyse the existing approaches for reducing its computational cost.

4.1.1 Mutation Testing applied to distributed systems

To the best of our knowledge, there are few proposals that applies MuT to distributed systems. In this line, Rutherford et al. used simulation to select the most effective test adequacy criteria and the most effective test suite among different adequate suites for the given criterion [155]. The authors apply traditional mutation operators to the simulation code, where all the

generated mutants are run against all the tests through an instrumented simulation. The authors used MuJava to generate mutants from the simulation code [114] and three distributed systems that are simulated in the SimJava simulation engine to evaluate the proposal [79].

Delamaro et. al presented a criterion for evaluating the adequacy of tests cases based on integration testing. This testing technique is based on checking the correctness of modular software, testing incrementally the different modules that compose the whole system. In that work, the authors propose *Interface Mutation*, a technique for assessing the quality of *how well* the interfaces that communicate different modules have been tested [55]. For this, the authors propose mutation operators to seed faults in the parts of the code that refer to interface communications, such as function calls, values returned from functions and global data sharing.

When compared with the proposal of Rutherford et al., the work in this thesis contributes to the field of application of mutation in a different way. In our framework, we propose the application of MuT techniques in simulation environments for checking simulation models. Hence, our approach is oriented to be used in simulation tools based on OMNeT++, which is written in C++. With respect to this programming language, none of the available MuT contributions designed for C++ are suitable for its application to our framework [107, 136]. Delgado-Pérez et al. [136] propose different class mutation operators for the object-oriented features of the C++ language. Although the proposed operators could fit into our proposal, it is well known that general mutation operators do not provide the same effectiveness in finding errors under a specific domain. Hence, we have designed a set of domain-specific operators explicitly adapted for distributed applications deployed on simulation platforms.

Kusano et al. [107] apply MuT to C/C++ concurrent programs. In this case, mutation is performed using their tool, CCMutator, which injects faults at concurrency constructs, such as semaphores, locks and mutual exclusion, among other mechanisms. However, we are interested in the mutation of calls to the methods included in the OMNeT++ API, as well as in the SIMCAN and MPI APIs. Therefore, we have defined a set of specific mutation operators that introduce faults affecting these calls. We have also developed a new mutation tool, MuTomVo, to implement the new framework. We are not aware of any other work in the context of simulation of distributed systems that integrates MuT techniques.

Since our mutation operators seed syntactical changes in function calls, the major part of them differs from the original *Interface Mutation* concept. First, the operators related to deletion, replacement and shift operations, are based on injecting errors that represent a lack of statements, wrong statements and wrong statement order. These errors affect the program behaviour without directly affecting the connection between modules. Second,

although some of the proposed mutation operators affect to the connections of the modules, such as signal swapping, time replacement and length replacement, all of them have been designed ad-hoc for each class of operators, representing a specific common error committed by competent programmers and gathered from diverse repositories and expert knowledge. On the contrary, the interface-based mutation operators proposed by the authors in their experiments [55] are designed for general purpose, based on C programming mistakes, but do not aim at reproducing typical errors in distributed systems.

4.1.2 Techniques for optimising mutation testing

It is well known that high computational power is required to speed-up the MuT process [148]. Thus, several cost reduction techniques to improve the execution time of MuT can be found in the literature [90, 174]. In general, these techniques are divided into three approaches: *do fewer*, *do faster* and *do smarter*. The proposals presented in this section focus on parallel testing, which is classified in the *do faster* approach. Although some works in this research field can be found in the literature during the last decades, it is worth mentioning that most of the proposals were introduced during the early nineties and in the last four years.

The first contribution in parallel MuT can be traced back to 1988 with the work of Mathur and Krauser [116]. In their approach, they proposed a novel technique to reduce execution costs using a vector processor. In this work, multiple mutants are simultaneously executed in a single processor using a sequence of vector instructions. Even though the approach greatly increases the computational performance of the MuT scheme, it is limited to mutants generated with the scalar variable replacement operator. Afterwards the authors extended their work with a high performance approach based on shared-memory, called mutation unification, to support several existing mutation operators [104, 149]. In their studies, the compilation was identified as a major bottleneck of the scheme. However, this issue can be alleviated by using current techniques that can be found in the literature [114, 173].

Currently, there exist multiple MuT frameworks that include parallel techniques to improve the performance and, consequently, to reduce the computational cost [89, 160]. Despite the benefits obtained by the use of Single Instruction Multiple Data improvements, these systems are limited by the number of available processors. Hence, it is necessary to include new distributed schemes of MuT that address this scalability issue.

In order to alleviate this problem, Offut et al. proposed the first MuT approach based on Multiple Instruction Multiple Data systems [133]. This work presents a parallel interpreter, called *HyperMothra*, which was implemented on a sixteen processor Intel iPSC/2 hypercube. In addition, diverse static schemes of distribution algorithms are included, such as distributing mu-

tants in original order and distributing mutants randomly and uniformly by mutation type. The authors stated that the obtained performance achieves almost a linear speedup over the Mothra sequential interpreter but they also identified the communications as the bottleneck of the system.

In the same line, Byoungju and Mathur presented the $P^M othra$ system [52]. This approach has a flexible architecture designed to provide a high degree of scalability. The system also provides the tester with a transparent interface to a distributed machine and includes a dynamic distribution algorithm that serves mutants to the available nodes. As in the previous proposals, the communication network is a bottleneck and slows down the performance of the system.

Most recently, Mateo and Usaola have presented a study for adapting the existing cost reduction techniques to current technologies [148]. They introduce $Bacterio^P$, a parallel extension of the MuT tool *Bacterio* [147], using *Java-RMI* [67] in order to communicate the nodes of the network. In addition, the authors include five distribution schemes using dynamic and static distributions. These schemes include the *Parallel Execution with Dynamic Ranking and Ordering (PEDRO)* algorithm. This is a dynamic distribution algorithm based on *Factoring Self-Schedulling* ideas [80], which considers addressing the well-known communication efficiency problem. Although this proposal achieves better results than the previous works, the mechanisms used in the communications are not the most adequate for high performance environments due to the high latency introduced by this technology. It has been shown that Java-RMI is 3 to 5 times slower than MPI [145]. Hence, we consider that the distribution process can be improved in order to achieve a higher level of parallelism by increasing resources efficiency.

In 2014, Saleh and Nagi presented the *HadoopMutator* framework. It is based on Map-Reduce programming model to distribute and execute the mutant generation and testing processes [156]. The framework is based on the *Hadoop* engine and the *Pitest* MuT framework. This approach follows a static schema in which the inclusion of dynamic distribution algorithms is not considered. Hence, this framework is not oriented to heterogeneous and dynamic environments.

Although these works aim at reducing the computational cost associated to the MuT process, they have some weaknesses, such as communication bottlenecks caused by, among others, the use of inappropriate technologies and suboptimal distribution algorithms. In order to alleviate these issues, in this thesis we provide a set of optimisations to improve the overall performance of the MuT process.

4.2 Mutation Testing framework for simulated cloud and distributed systems

Testing distributed systems may be an arduous and complex task. There are several factors that hamper the testing process like, among others, the execution of the test cases against the target system and the need for exclusive access to this kind of systems. When the utilisation of the system under test is not exclusive, the execution of applications launched by other users may also affect the testing process, causing unfavourable situations such as bottlenecks or delays in the communication network. This fact is not acceptable when testing non-functional characteristics such as performance or energy consumption. In order to overcome these problems we have developed a flexible and adaptable framework, called MuTomVo, to carry out the testing process in simulated environments, where any application that includes API calls or external libraries can be tested.

MuTomVo is a MuT framework that provides mechanisms to generate and evaluate the effectiveness of test suites to check distributed systems. In essence, MuTomVo applies mutation operators for reproducing the common mistakes, made by competent programmers developing distributed applications. The main components of MuTomVo, the testing process and the method for automatic tests generation, are described along this section. It is important to emphasise that the evaluation process carried out to determine the applicability of MuTomVo can be found in the paper associated to this contribution [39] (7.3).

4.2.1 Architectural design

The architecture of MuTomVo is illustrated in Figure 4.1. This scheme depicts how MuT techniques are integrated with modelling and simulation tools. In addition, the diagram shows the required steps to perform the testing process. The main goal of this process is to estimate the effectiveness of a test suite to detect errors in applications executed over simulated distributed environments.

Initially, the user must build a system model using the *GUI* of the SIMCAN simulator [127]. This model consists of two main components. The first one, known as *simulation scenario*, is the configuration of the distributed architecture, including physical machines and network connections, which are used by the simulation platform to deploy the system. The second one is the application model that will be executed over the provided architecture. It should be noted that these application models are an abstraction of real applications, written in C++, which contain its most relevant patterns and characteristics, like CPU processing, message passing, I/O and network throughput, among others. In order to facilitate the design process, SIM-

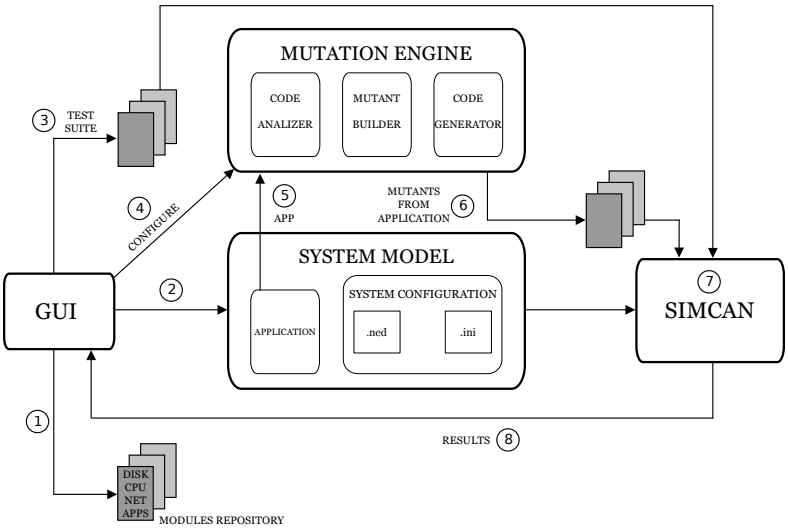


Figure 4.1: Architecture of MuTomVo.

CAN offers a repository of predefined modules and a set of applications that can be used by the user to design its own model ①. These modules simulate the behaviour of the elements, such as CPUs, disks, memories and communication networks that compose the distributed system. The applications that are included in the repository cover different paradigms of data processing and communications, such as Map-Reduce, client-server and single node computation [127]. In addition, new applications can be modelled by the user using the APIs provided by the simulation platform. Once the design of the model is complete, MuTomVo generates the configuration files required by SIMCAN to simulate the designed environment ②. In step ③ a test suite for testing the application model is provided. The test cases can be either created manually by the user or generated automatically by MuTomVo. Next, the mutation engine must be configured ④. The user must establish the required information to perform the mutation process. This information corresponds to the application designed in ②, the test suite built in ③ and the mutation operators that will be applied to the application model for generating the mutants. Once the configuration phase is completed, the mutation engine receives the application and starts the mutant generation process ⑤. This way, the mutation engine applies the selected mutation operators to inject different faults in the original code of the provided application model for creating faulty versions, that is, mutants ⑥. At this point, the framework has the system model, the application and the generated mutants. SIMCAN uses the system model ⑦ to build the architecture

and topology configured at step ①. Next, all the mutants are compiled and both the original application model and the mutants are executed against the test suite in SIMCAN. The results obtained from the execution of the original application model are compared with the ones obtained from each mutant execution in order to check if the mutants are killed or alive. Finally, the results are sent to the GUI module to present them to the user ⑧.

4.2.2 Mutation engine

The continuous development of new contributions in the MuT field requires the availability of a flexible environment that can be extended with new operators and optimisation techniques. For this purpose, MuTomVo was built using a modular and flexible design.

On the one hand, MuTomVo is modular in the sense that its functionality is divided into independent modules. Figure 4.2 shows the four modules used to carry out mutation analysis: *mutation engine, code analyser, mutant builder* and *code generator*. Consequently, different modifications can be applied to each module without interfering with the rest of the framework.

On the other hand, our proposed framework is flexible in the sense that different approaches can be integrated into each module. These features easily allow the inclusion of both existing and new techniques, such as selective mutation [137], trivial compiler equivalence [141] and mutation clustering [113]. This aspect significantly reduces the integration time and increases the feasibility of having a high number of techniques in the framework.

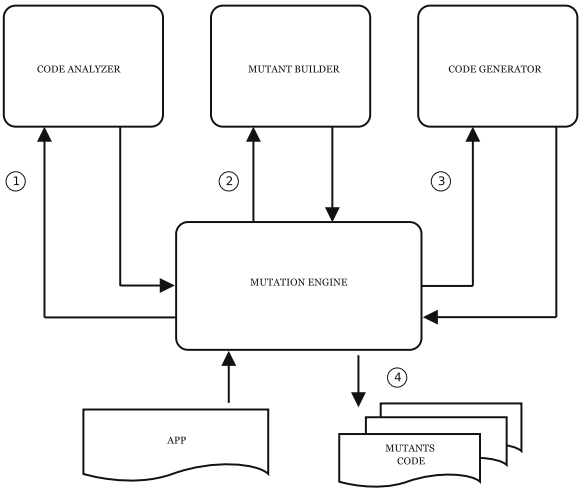


Figure 4.2: Architecture of the mutation engine.

The main element, the *mutation engine*, is responsible for orchestrating the communication among the other components of the framework. This module provides a high level of flexibility, supporting the inclusion of new

mutation operators easily. When the mutation process starts, the *mutation engine* receives as input the application to be mutated and transfers it to the *code analyser* ①. The *code analyser* module analyses the source code in order to determine possible mutation operators targets. With this goal, the Eclipse C/C++ development tooling parser (in short, CDT) [143] has been integrated into the tool. The CDT parser provides mechanisms to facilitate the code analysis, such as an abstract syntax tree, which avoids the user building its own syntactical analyser. In addition, CDT has an open-source license and is actively maintained by the Eclipse community. All of that makes CDT a robust and major asset for the active and flexible development of MuTomVo. The data structures generated by the code analyser are provided to the *mutant builder* ②, which produces the mutants and saves them in memory. Let us remark that the mutant builder does not generate syntactically invalid mutants. It takes into account several factors depending on the mutation operator applied, such as the data type of the parameters required to invoke the methods affected by the *shuffle parameters* or the *replace* operators. Finally, the *code generator* creates and stores in disk the source code obtained in the previous step ③. This module is based on a layered architecture where the first layer implements basic operations, such as exporting the source code to disk or mutant enumeration, and the second layer extends it by using specific source code generators: a standard code generation and an API-based code generation.

4.2.3 Tests cases

In this framework, a test case corresponds to a list of pairs $< parameter, value >$ that configures the execution of the application model and its mutants, such as the amount of computation (measured in millions of instructions) and the size of processed data (measured in GB). The structure of the test cases for each specific application model is provided by the user. It must contain all the necessary parameters for the simulation of the application model. All the test cases will assign a value to each one of the input parameters.

For example, let us consider an application, called *appCpu*, which performs different operations over a data set in a loop. Basically, while the data set is not processed entirely, the application reads a piece of data, performs computation and writes the result to a file. Listing 3 shows a test case for this application model. All the required parameters are properly set a value, as needed for simulating the execution of the application and the generate the mutants. In this case the parameters denote the size of the data set to be processed (*inputDataSize*), the size of the file that stores the results (*outputDataSize*), the computation of a piece of data in millions of instructions

```
1 int inputDataSize=19MiB;   // Size of data-set
2 int outputDataSize=8MiB;   // Size of results
3 int MIs=698667;      // Computing for each iteration
4 int iterations=5;    // Number of iterations
```

Listing 3: Example of test case.

(*MIs*) and the number of times the application will be executed (*iterations*).

In addition to the structure of the test cases, which only consists of the input parameters, the tester also needs to provide the system with the output parameters that must be returned by the simulation of the application model and the mutants. The values returned by the execution of the original application model against the test cases will be used as *oracle*. If the values returned by the execution of the mutant are not equal to the ones produced by the original application model, the mutant is killed, otherwise, it is alive.

In our example, the output parameters correspond to the time spent in input/output operations during the execution (*ioTime*), the CPU time (*cpuTime*) and the simulation time (*simTime*). Listings 4 and 5 show the values returned by the execution of the application and by a mutant against the previous test case, respectively. In this case, the difference between the values indicates that the mutant has been killed by the test case.

If either the values of the output parameter returned by a mutant does not correspond to the ones produced by the original application model or the execution time exceeds the established timeout, the mutant is not executed against the rest of the test cases and it is classified as killed.

```
1 ioTime=14.857143;
2 cpuTime=0.720575;
3 simTime=15.577717;
```

Listing 4: Program output.

```
ioTime=10.785714; ◄
cpuTime=0.330697; ◄
simTime=11.116411;◄
```

Listing 5: Mutant output.

The generation of test cases can be manually done by using the MuTomVo GUI. However, providing a test suite that comprehensively checks an application model is a very expensive and error-prone task. In order to overcome these difficulties, we have developed a method to automatically generate test suites using a random approach. The configuration parameters included in the structure of the test cases are used to create a collection of instances. In addition, the user must select the set of parameters that will be randomly valued, the maximum and minimum values that each parameter can take and the total number of tests to be generated.

4.2.4 Testing process

Figure 4.3 illustrates the generation of mutants and the execution process. Initially, the process starts when the user provides MuTomVo with three elements: a system architecture model, an application model to be executed in this architecture and a test suite for checking the correctness of the application model. For the sake of simplicity, the system architecture and the application model are represented as *System model* in the diagram.

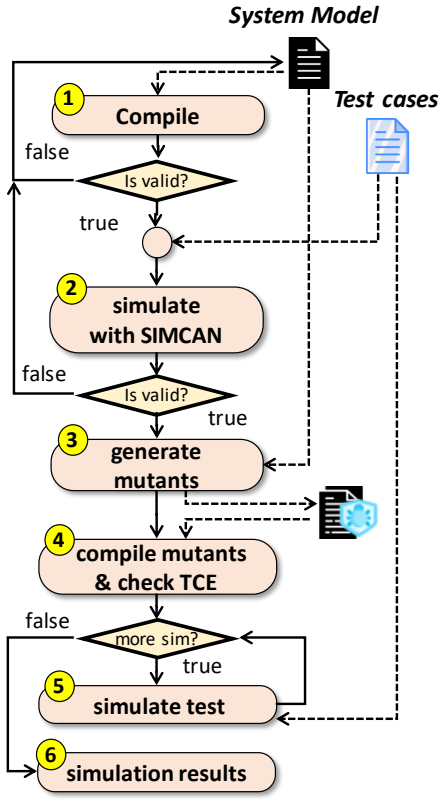


Figure 4.3: Testing process.

The first task of the testing process is the compilation of the application model, carried out by the GCC compiler ①. Once the compilation phase has finished, the execution of the original application model against each test case is simulated in SIMCAN ②. In order to ensure the validity of the provided system model, the produced outputs are checked; in case that the output reveals a configuration error, both the application and the architecture must be revised to fix the system model. There are different types of errors that can be identified in a simulated environment, which are, among others:

- Architecture misconfiguration. The topology of the distributed system is not well configured, being some of the components disconnected from

the communication network.

- The application does not finish its execution. In this case, the user must manually set a timeout to stop the simulation. Since detecting whether a program will stop or not is an undecidable problem (see the halting problem [172]), the user must know the application under test and then set a pre-defined timeout depending on the values used in the test case.
- The application accesses a file that does not exist in the file system of the simulated environment.
- The application does not allocate enough memory. Then, when the user writes some information in the memory, an exception is thrown.
- Occurrence of unhandled exceptions. The application throws unmanaged exceptions, like division by zero, which are not captured by the user.

On the contrary, if none of the errors described above occurs, the tool will proceed with the generation of the mutants ③. Let us remark that MuTomVo has been designed to generate valid mutants in the sense of producing syntactically correct programs to be successfully compiled.

Then, the mutants are compiled and analysed to check whether any of them are equivalent ④. In order to automatically identify both equivalent and duplicated mutants, the Trivial Compiler Equivalence (in short, TCE) approach has been included in the testing process [141]. A mutant is considered equivalent when it is semantically identical to the original program and there is no test case that can kill it. Similarly, duplicated mutants are considered a special form of equivalent mutants in a way that they are equivalent to each another, but not to the original program. Basically, TCE detects equivalent mutants by compiling each mutant and comparing its executable file with the executable file of the original program. We assume that a mutant is equivalent when both executable files are identical. Similarly, TCE also detects duplicated mutants by comparing executable files generated from different mutants. Hence, both equivalent and duplicated mutants are removed from the generated mutant set.

After the mutant compilation, all of them are executed sequentially against each test case ⑤. If the values of the output parameters returned by a mutant do not correspond to the ones produced by the original application model or the execution time exceeds the established timeout, the mutant is not executed against the rest of the test cases and it is classified as *killed*. Otherwise, if all the outputs produced by the execution of a mutant against all the test cases are equal to the ones produced by the application, it is said that the mutant is *alive*. At the end of this process, the results are shown to the user through the MuTomVo GUI ⑥.

4.2.5 Experiments and validation

This section describes the evaluation process carried out to determine the applicability of MuTomVo. The main goal of these experiments is to analyse the effectiveness of test suites for detecting errors in distributed applications.

The selected applications to carry out the experiments cover three well-known paradigms used in both cloud and HPC infrastructures, these are, client/server paradigm, scientific applications that mix CPU-intensive and data-intensive paradigms, and the pipeline paradigm [86, 110, 157].

These applications have been developed and deployed in three different simulated data-centres. For each one of them, we have generated both a specific test suite to check the applications and a set of mutants as a result of applying the proposed mutation operators, which can be consulted in the paper [39](7.3). The test suites have been randomly generated. All the test suites contain 100 test cases. The application of the proposed mutation operators to the applications has produced 514 mutants that have been executed against the tests cases, which means a total of 51,400 simulations. This engine uses an abstract syntax tree that aids in the injection of errors in such a way that only syntactically correct mutants are generated.

The effectiveness of a test suite is measured on the basis of the *mutation score* (in short, MS), that is, the percentage of non-equivalent mutants that the test cases have killed. We have classified the results by the mutation operator that generated the mutants and the total time required to execute each test case. The latter criterion will help us to determine the test cases that kill the maximum number of mutants but spend the minimum time in the testing process.

In the experiments, 102 equivalent mutants have been found. Initially, we use TCE to automatically detect equivalent mutants. However, this technique was not able to identify all of them and we had to do it manually. The time invested to analyse and identify each of them did not take more than 3 minutes. The highest number of equivalent mutants was generated by the OMNeT++ operators due to their reduced impact on the behaviour of the applications.

All the experiments were performed on a 8-node cluster, where each node is provided by a Quad-Core Intel(R) Core(R) i5-3470 CPU at 3.4 Ghz with hyper-threading, 8 GB of RAM and 500GB HDD. These nodes are interconnected through an Ethernet Gigabit network.

The proposed framework has been analysed by performing the testing process using one test suite and two mutant sets. The first mutant set is generated by using the proposed mutation operators (see Table 4.1) and the second one contains mutants generated by using traditional mutation operators (see Table 4.2). It is important to remark that the same test suite has been used for both mutant sets.

Operator	Mutants				MS(%)
	Generated	Alive	Killed	Equivalent	
OMCD	30	6	16	8	72
OOPD	31	0	29	2	100
OMCR	16	0	9	7	100
OOMU	25	0	8	17	100
OOMD	23	0	2	21	100
OOSS	4	0	4	0	100
OOSNR	5	0	5	0	100
OOSTR	57	0	39	18	100
OMNeT++ operators	**191**	**6**	**112**	**73**	**95**
SMCD	26	2	24	0	92
SMCR	25	0	25	0	100
SOMU	25	0	18	7	100
SOMD	14	0	6	8	100
SOSP	9	0	9	0	100
SOFNM	14	0	14	0	100
SORDMA	1	0	1	0	100
SOCIR	4	0	4	0	100
SIMCAN operators	**118**	**2**	**101**	**15**	**98**
MMCD	29	0	29	0	100
MMCR	56	0	56	0	100
MOMU	23	0	18	5	100
MOMD	27	0	20	7	100
MOSP	14	0	14	0	100
MOPIR	28	1	26	1	95
MOBLR	28	1	26	1	95
MPI operators	**205**	**2**	**189**	**14**	**99**
Total	**514**	**10**	**402**	**102**	**97.5**

Table 4.1: Results of applying MuT in all the applications

Regarding to the effectiveness of the mutation operators, a total number of 4019 mutants have been generated by using traditional operators, while only 514 mutants have been generated when our proposed mutation operators are applied. In this study, 1683 equivalent mutants (41.87%) have been generated by applying traditional mutation operators, while the mutant set generated by using our proposed operators only contains 102 equivalents (19.82%). In this latter case, TCE techniques have not detected duplicated mutants, which indicates the high quality of the generated mutant set. However, it is interesting to highlight that SIMCAN and MPI operators only produce 15 (12.71%) and 14 (6.82%) equivalent mutants, respectively. Hence, the proposed mutation operators create a reduced set of mutants that

has a high impact in the control flow of the application, which represents a substantial profit, both in terms of computational resources and time costs, in comparison with the traditional mutation operators that generate a vast number of mutants. This fact addresses one of the challenges of mutation testing, which is considered as a computational expensive technique. This solution is suitable to perform MuT over distributed applications in simulated environments.

Operator	**Mutants**				**MS(%)**
	Generated	**Alive**	**Killed**	**Equivalent**	
AORb	392	26	234	132	90
AORs	72	33	39	0	54.1
AOIu	796	96	353	347	78.6
AODs	72	29	43	0	59.7
ROR	1570	116	677	777	85.5
LOI	796	81	417	298	83.7
COR	138	20	55	63	73.3
COI	92	13	58	21	81.6
COD	27	2	23	2	92
ASOR	64	8	28	28	77.7
Total	**4019**	**425**	**1911**	**1683**	**81.8**

Table 4.2: Results of traditional operators

Regarding to the suitability of the obtained MS, it can be seen that the testing process using a mutant set generated by applying the proposed mutation operators provides a better MS, on average (97.5%), than the testing process using a mutant set generated by using traditional operators (81.8%). Additionally, we observe that, in the major part of the cases, we obtain a MS of 100 when our proposed operators are used. On the contrary, traditional operators provide poor results, which in this study ranges from 54.1 to 92. Then, the testing process provides promising results, in terms of MS, when a high-quality mutant set is used during the testing process. Since the obtained MS reaches 100 in the major part of the analysed mutant set, we can state that this approach is appropriate to be applied in MuT. Moreover, we observe a correlation between the number of generated equivalents and the MS. Broadly speaking, a test suite executed over a mutant set generated by using a mutation operator that produces a reduced number of equivalents obtains a better MS that the same test suite executed over a mutant set generated by applying a mutation operator that generates a high number of equivalents.

4.3 Techniques for improving the performance of mutation testing

MuT is a computationally expensive testing technique, since the number of mutants that are generated is huge and they must be executed against the test suite. In consequence, high computational power is required to speed-up the MuT process. Nowadays, there are several techniques to improve the performance of this testing process [90, 174]. In this section, we describe several optimisations, bridging the gap between one of the main limitations of MuT, its high computational cost, and the main advantages provided by HPC systems, parallel infrastructures to speed-up the execution of computational applications. The main goal of these techniques is to achieve an *scalable*, *dynamic* and *high performance* solution to face the computational challenges associated with MuT.

- *Scalable*: The proposed techniques have been designed to be deployed and executed in a distributed system. The increment in the quantity and quality of system resources means the increase in its computational performance. It presents two types of scaling: *horizontal and vertical*. The former allows to include more computing nodes to the system, while the latter allows extending the computational resources in each node.

- *Dynamic*: In order to maximise the exploitation of computational resources, it splits the input dataset into blocks and dynamically delivers them to the available CPUs. Once a process finishes the execution of a block, it is provided with a new block until all the blocks have been processed. This distribution scheme benefits heterogeneous systems.

- *High performance*: The proposed algorithm is based on a *high performance* schema in which the shared resources of several machines are used as a whole to perform the MuT process. The testing process is executed in parallel over all the machines of a cluster, taking advantage of the low latency communication network to maximise the parallelism and enhance the overall performance.

In the following, Section 4.3.1 and Section 4.3.2 present the proposed improvements to speed-up the MuT process. The former describes an algorithm for distributing the workload of the MuT process. The latter describes a set of 4 optimisation strategies, which are based on the previously presented algorithm, to improve the overall performance of the MuT process.

4.3.1 EMINENT: EMbarrasINgly parallEl mutatioN Testing

In this section we describe EMINENT, an algorithm to distribute the workload of the MuT process for reducing its execution time. In order to analyse the suitability of the algorithm, it has been implemented within a generic framework, which allows to apply parallel MuT regardless of the mutation framework used. Next, we describe its main features.

The proposed scheme uses different processes. On the one hand, the *master* process is responsible for orchestrating the algorithm. It splits the workload of the testing process in *execution blocks* and distributes them among the *worker* processes. On the other hand, the *worker* processes execute them and send the results back to the *master* process. The number of *workers* processes that are instantiated in the algorithm is variable and can be defined by the user.

Figure 4.4 shows the basic scheme of EMINENT. The first step consists of the selection of both the *source code* of the program to which the MuT process will be applied and the *test suite* that will be used during this process. Then, the *master* process compiles the original program ② and, if the compilation finishes successfully, the *testing process* begins. At this point, the *master* executes all the test cases in the selected test suite and stores the results ③. If the execution of all test cases is correct, the *master* invokes an external MuT tool to generate ④ and compile ⑤ all the program mutants. Mutants are produced by using *mutation operators* that aim to simulate common faults. Each mutation operator makes a small syntactic change in the source code. The execution of the generated mutants is distributed by the *master* process among the *workers* ⑥. For each mutant, the *master* process dispatches the test cases to the *worker* processes that will execute them against the mutant. The obtained results are sent to the *master*, which compares them with the ones produced by the original program. In the case that a difference is detected, the mutant will be considered *killed*, and all the running executions associated with it will be aborted and no more tests will be executed against it. The process continues until all the test cases are executed against all the mutants. Finally, the *master* process calculates the *mutation score* of the process, which indicates the percentage of killed mutants over the total number of mutants.

The experimental phase conducted to validate EMINENT can be found in the paper associated with this contribution [31] (7.4).

4.3.2 OUTRIDER Optimising the mUtation Testing pRocess In Distributed EnviRonments

In this section we describe OUTRIDER an HPC-based optimisation of the MuT process, which uses the EMINENT algorithm as the basis. Since EMINENT does not properly exploit the resource usage in HPC systems, this thesis

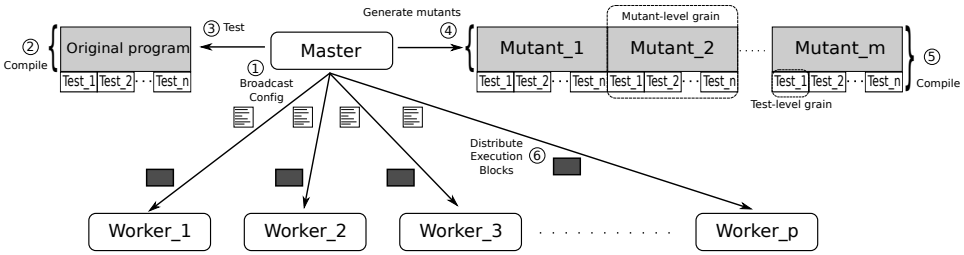


Figure 4.4: General scheme of `EMINENT`.

proposes an optimisation consisting of 4 different strategies to alleviate this issue. It is worth noting that the examples used along this section are illustrative. An exhaustive experimental phase, which shows the adequacy of the proposed optimisations, can be found in the paper associated with this contribution [37] (7.5).

4.3.2.1 Parallelising the execution of the test suite over the original application

Usually, the sequential execution of a test suite over the original program is an issue that hampers the scalability of the MuT process [148]. This becomes especially relevant in those cases where the application under test requires a long execution time and when the test suite consists of a large number of test cases. Consequently, the scalability of the system is compromised due to the lack of parallelism, which is generally reflected in a low system performance.

In order to alleviate this issue, we propose exploiting the resources of the system by executing – in parallel – the test suite over the original program. Basically, this strategy consists of distributing the execution of each test over the original program among different processes, which are executed in the available CPU cores of the system.

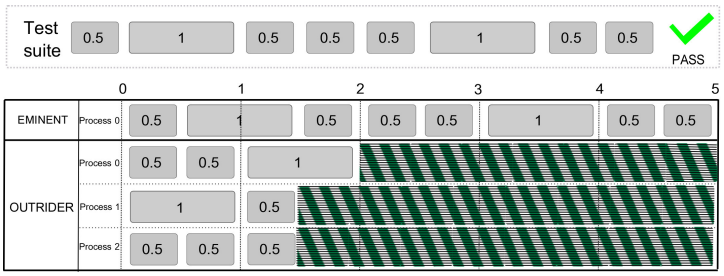


Figure 4.5: Execution of a test suite over the original program using `EMINENT` and `OUTRIDER` .

Figure 4.5 presents an example comparing the execution of a test suite over the original program using `EMINENT` and `OUTRIDER`. The schema at the

top shows the sequential execution of the test suite, where each rectangle represents a test case and the number inside it shows the time slots required for its execution. The test suite consists of 8 test cases and has a total duration of 5 time slots. While the test suite is sequentially executed in one processor using `EMINENT`, `OUTRIDER` parallelises the execution of the test suite using 3 processes. Notably, this example shows that the distribution of the test suite improves the overall performance, obtaining a speed-up of 2.5.

4.3.2.2 Sorting the test suite

In MuT, test cases are executed over a mutant until the mutant is killed or the test suite is completely executed, and the mutant remains alive. It is therefore desirable that the mutant gets killed as soon as possible. Unfortunately, we cannot determine which test cases will kill the mutant before executing them. However, we can use information gathered from the execution of the test suite over the original program, like the execution time of each test case.

This strategy uses this information to specify the execution order of the test cases. Thus, test cases are sorted by using its execution time as sorting criteria. As a result, the fastest test case is processed in the first place, while the slowest test case is executed last. The idea is to minimise the required time to kill a mutant. Although sorting the test suite has a computational cost, we assume that applying this strategy would reduce the overall execution time.

As an example, Table 4.3 shows the execution of a test suite with 5 test cases over a mutant. The first two columns, TC_{EMI} and TC_{OUT}, refer to the order of execution for each test case using `EMINENT` and `OUTRIDER`, respectively. The next two columns, ExecTime_{EMI} and ExecTime_{OUT}, represent the execution time for each single test case. These are followed by the two columns that represent the cumulative time required to execute the test cases using `EMINENT` and `OUTRIDER`. The last column shows the improvement obtained by comparing `OUTRIDER` and `EMINENT`, where positive values indicate that `OUTRIDER` executes faster than `EMINENT` and negative values show otherwise. This improvement is calculated by taking into account that there is a test case that kills the mutant. In this example, `OUTRIDER` obtains better results than `EMINENT` when the test case 2, 3 or 5 kills the mutant, obtaining an improvement in the total execution time of 1175, 964 and 3144 seconds, respectively. On the contrary, `EMINENT` executes faster than `OUTRIDER` when test case 1 or 4 kills the mutant.

4.3.2.3 Enhancing the test case distribution strategy

In order to increase both the level of parallelism and the resource usage efficiency, we propose a strategy that improves the workload distribution

$\mathbf{TC}_{EMI}$	$\mathbf{TC}_{OUT}$	$\mathbf{ExecTime}_{EMI}$	$\mathbf{ExecTime}_{OUT}$	$\mathbf{Acc}_{EMI}$	$\mathbf{Acc}_{OUT}$	Improv.
1	2	1175	139	1175	139	-848
2	5	139	211	1314	350	1175
3	3	498	498	1812	848	964
4	1	1471	1175	3283	2023	-211
5	4	211	1471	3494	3494	3144

Table 4.3: Execution time, in seconds, of 5 test cases over a mutant using `EMINENT` and `OUTRIDER`.

presented in `EMINENT`, which additionally considers the number of remaining mutants to be completely executed, the number of processes involved in the testing process and the number of processes that are executing each mutant. The idea is to improve the resource usage by maximising the number of different mutants executed in parallel. Thus, when the number of remaining mutants to be executed is greater than or equal to the number of available processes, each single process executes a different mutant. Otherwise, the remaining mutants to be completely processed are proportionally distributed among the available processes.

Figure 4.6 shows an example that compares two different distribution strategies. In this example, a test suite consisting of 3 test cases is executed over 4 mutants using 2 processes, each one having a dedicated CPU core. The schema at the top depicts the execution of the test suite over each mutant, where test case 1 kills mutant 1 and 4, test case 2 kills mutant 3, and mutant 2 remains alive. The schema at the bottom shows the strategies used by `EMINENT` and `OUTRIDER` to distribute the workload in the MuT process. The execution of the test case Y over the mutant X is denoted by mX.Y. In this scenario, executions m1.1, m3.2 and m4.1 kill the processed mutant.

The distribution strategy used in `EMINENT` shows that the execution of some test cases is useless. For instance, m1.1 and m1.2 are executed in parallel. Although the former execution kills mutant 1, process 1 is wasting computational resources by executing m1.2, which is not necessary to kill the mutant. Since the strategy used in `OUTRIDER` maximises the number of different mutants executed in parallel, this situation is avoided in most scenarios. In this example, `OUTRIDER` obtains an improvement of 20% in the total execution time.

4.3.2.4 Categorising equivalent mutants using Trivial Compiler Equivalence

As we introduced in Section 4.2.4, in MuT, a mutant is considered equivalent to the original program when it can not be distinguished from it through testing. The equivalence problem is one of the main obstacles in the practical use of MuT. Although it is well known that deciding whether two programs are equivalent is a non-decidable problem [132], there are several heuristics

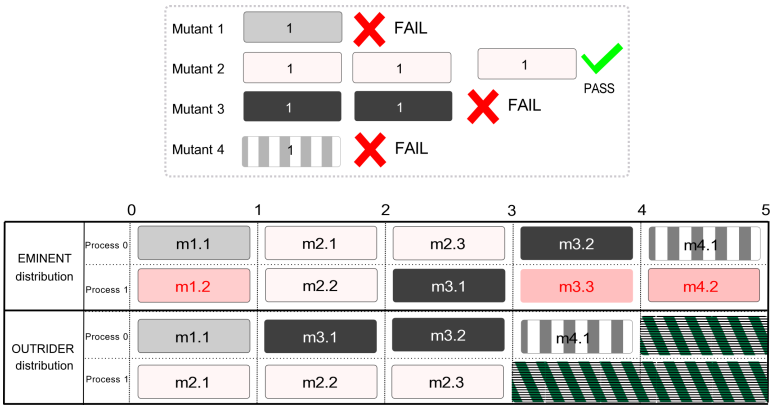


Figure 4.6: Workload distribution using `EMINENT` and `OUTRIDER`.

that aid in finding patterns to identify this kind of mutants. In this case, due to its simplicity and its computational efficiency, we have selected the TCE method to detect both equivalent and cloned mutants [141]. This technique uses compiler optimisations to detect patterns that help to identify the equivalence between programs using a black-box scheme.

Using this strategy we detect two kinds of mutants. On the one hand, mutants that are equivalent to the original program, known as equivalents. On the other hand, mutants that differ from the original program but are equal to other mutants, called cloned mutants.

We apply this technique after the compilation phase, where both equivalent and cloned mutants are detected. Mutants identified as equivalents are discarded and none of them is executed. On the contrary, cloned mutants are grouped in domains, where a single mutant is selected as *representative* of the domain. During the testing phase, only those mutants that do not belong to a domain are executed, which are handled as usual. Next, for each domain, only representative mutants are processed. Once the execution of a representative mutant ends, if the mutant is killed, only the killer test is applied to the rest of the mutants of the domain, which substantially reduces the number of test case executions. On the contrary, if the representative mutant remains alive, the rest of the mutants of the domain are managed as usual.

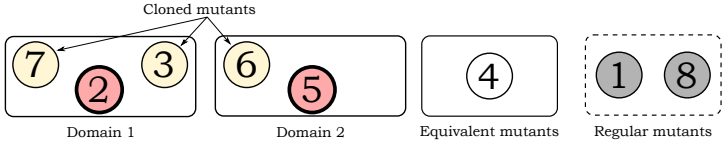


Figure 4.7: Categorisation of cloned and equivalent mutants.

Figure 4.7 shows an example that illustrates the execution of a MuT

process with 8 mutants. We applied our strategy to categorise these mutants, obtaining two different domains. Domain 1 consists of mutants 2, 3 and 7, and Domain 2 consists of mutants 5 and 6. Mutants with a bold border are the representatives of its domain (mutant 2 for Domain 1, and mutant 5 for Domain 2). Mutant 4 has been detected to be equivalent, while mutants 1 and 8 are categorised as regular mutants.

Mutant ID	**Exec.Time**	**Time**$_{EMI}$	**Killer test time**	**Time**$_{OUT}$
1	432	432	-	432
2	245	677	-	677
3	245	922	46	723
4	456	1378	-	723
5	532	1910	-	1255
6	532	2442	164	1419
7	245	2687	46	1465
8	591	**3278**	-	**2056**

Table 4.4: Execution of 8 mutants using `EMINENT` and `OUTRIDER` with TCE.

Following the example, Table 4.4 presents the execution time of the MuT process. The first two columns, *Mutant ID* and *Exec.Time*, represent the mutant ID and its execution time, respectively. $Time_{EMI}$ refers to the accumulated time when the testing process is executed using `EMINENT`. The next column refers to the execution time of the test that kills the mutant, which is calculated from the representative mutant of each domain. Finally, $Time_{OUT}$ refers to the accumulated time when the testing process is executed using `OUTRIDER`.

These results show that `OUTRIDER` executes 37% faster than `EMINENT`. That is, while `EMINENT` requires 3278 seconds to completely execute the testing process, `OUTRIDER` requires 2056 seconds. This performance improvement is obtained because `OUTRIDER` executes fewer test cases than `EMINENT`. In this case, mutant 3, 6 and 7 are not completely executed because only the test case that kills them is executed instead.

4.4 Summary and conclusions

This chapter has presented the contributions made in the field of MuT. In this thesis, we use MuT for improving the proposed methodologies based on MeT, which has been described in Section 3. Moreover, we apply MuT to analyse the effectiveness of the MRs in finding errors. For this, several syntactic faults are injected using the mutation operators, on both the software parts of the cloud and the applications executed on them. The latter corresponds with the faults seeded by using the mutation operators designed in MuTomVo.

In addition, we use MuT for generating suitable test suites for testing the distributed applications executed over the cloud. Finally, the proposed

optimisation strategies are used to reduce the computational cost associated to the execution of these test suites.

4.5 Associated publications

(7.3) **Mutomvo: Mutation testing framework for simulated cloud and HPC environments.**
Pablo C. Cañizares, Alberto Núñez and Mercedes G. Merayo.
Journal of Systems and Software 143: pp. 187-207, 2018.

(7.4) **EMINENT: EMbarrassINgly parallEl mutatioN Testing.**
Pablo C. Cañizares, Mercedes G. Merayo and Alberto Núñez.
In Proceedings of 16^{th} International Conference on Computational Science. pp. 63-73, 2016.

(7.5) **OUTRIDER: Optimizing the mUtation Testing pRocess In Distributed EnviRonments.**
Pablo C. Cañizares, Alberto Núñez and Juan de Lara.
In Proceedings of 17^{th} International Conference on Computational Science. pp. 505-514, 2017.

Chapter 5

Modelling and optimisation of cloud systems based on structural rules

Cache rules everything around me
Word up, you gotta get over.

Method Man.

This chapter presents some relevant contributions in the field of MDE applied to modelling cloud systems. The main goal is to use MDE to enable reasoning and optimization of cloud structures using expert rules. Whereas in this section it is done statically, dynamic techniques have been introduced in chapters 3 and 4. Section 5.1 presents a study of the existing modelling languages related to cloud applications and systems. Section 5.2 describes the proposed DSL to design the data-centres that support cloud infrastructures. Section 5.3 provides several expert rules for detecting and suggesting changes to fix possible misconfigurations. Finally, a summary of the chapter and some conclusions are provided in Section 5.5.

5.1 State of the art

Currently, MDE has been succesfully applied to a wide spectrum of fields [35, 82]. In particular, there are many works in the literature that use MDE to face the different challenges that can be found in cloud computing, such as developing and deploying large scale and distributed cloud applications, representing all the components of cloud environments, and helping to solve the lack of interoperability between the existing cloud solutions [16]. This section studies some of the most relevant proposals to alleviate these issues.

5.1.1 Cloud modelling languages

In order to aid users in the design process of data-centres, several languages for modelling and analysing cloud systems have been proposed in the literature during the last years.

Frey and Hasselbring presented a model-based approach for migrating software systems to cloud-based applications. This proposal, called CloudMIG [66], equips SaaS providers with a method to migrate existing software to PaaS and IaaS applications. In particular, it is based on migrating applications to the cloud, with special interest in achieving optimal deployment configurations and their conformance with target cloud environments.

The topology and orchestration specification for cloud applications (in short, TOSCA) is an OASIS standard, setting the guidelines for representing portable cloud applications [21]. TOSCA provides an XML-based modelling language that represents the application structure as a graph and captures the management tasks in plans. TOSCA has three main objectives: automated deployment and management of applications, migration of applications and component reusability.

Silva et al. introduced Cloud DSL, a language that supports cloud portability by cloud entities covering a broad spectrum of cloud IaaS services [167]. This language is more descriptive and expressive than existing ones, since it facilitates cloud portability, the communication between services and resources and allows to model different cloud systems.

Rossini et al. presented CAMEL, an MDE approach for modelling and executing multi-cloud applications. This language allows modelling several aspects of cross-cloud applications, such as cloud security, tenants, application execution, service level, provisioning and deployment, among others. In addition, it supports models@run-time [22] to facilitates reasoning and adaptation of multi-cloud [152] and integrates and extends existing DSLs like CloudML [17], Saloon [144] and SRL [105].

Ferry et al. presented CloudMF, an approach that provides a DSL for specifying the provisioning and deployment of multi-cloud applications, and a models@run-time environment for their continuous provisioning, deployment, and adaptation [64]. Thus, CloudMF aims at unifying development and operation activities and provides different levels of control, depending on the infrastructure type where the application is executed. If the application is executed over a white box PaaS solution, it provides full control with automatic provisioning and deployment. On the contrary, if the application is executed over a black box PaaS solution, it provides shared control of the application.

Guillen et al. presented MULTICLAPP, a UML profile for modelling cloud applications from a cloud-provider independent viewpoint [69]. MULTICLAPP suports designing the components of the application as a software

artefact composition. Then, it can synthesize the application source code, for target cloud environments, from the previously designed components.

Andrikopoulos et al. presented a generalised topology language, called GENTL, which enables the description of deployment configurations with emphasis on cost-efficient application provisioning [8]. This language provides design support capabilities and facilitates the transformation of other topology models into a standard model. GENTL consists of two different subsystems: a knowledge base and an application topology language. The knowledge base contains information about the existing cloud providers, aids in selecting the most appropriate cloud and allows calculating the cost of a provided usage profile. The application topology language selects the optimal distribution of application components from a target cloud.

Benson et al. presented CloudNaaS [15], a cloud networking platform for deployment applications with special emphasis on networking aspects. For that purpose, CloudNaaS provides rich and extensible network services, such as fine-grained access control, VLAN-based isolation, middlebox interposition and service differentiation. In terms of performance, CloudNaaS is highly efficient since its primitives are directly implemented in the target cloud infrastructure. The performed experiments show that CloudNaaS performs well handling a large number of provisioning requests.

Hamdaqa and Tahvildari introduced StratusML, a modelling language for cloud applications that is able to generate executable deployment descriptors and runtime adaptation rules [70]. In order to facilitate the definition, configuration and cost estimation of a service, it provides an intuitive GUI. In addition, the user interface also allows to model and specify applications at runtime.

Holmes presented a proposal to facilitate the provisioning and development of service topologies using DSLs [77]. The main goal of the approach is to provide mechanisms supporting the description of deployment configurations and their automated provisioning.

Although these works are a suitable solution from the view point of modelling cloud services, the goal of these proposals differs from the main objective of this thesis. The goal of this part of the thesis is to provide a formal and detailed model of the cloud, representing its different component accurately, so that expert rules for optimising its design can be defined.

5.1.2 Model Driven Engineering frameworks for modelling and analysing cloud systems

There are several MDE-based frameworks for aiding users to model and analyse cloud systems. Palyart et al. presented MDE4HPC [140], a model-based approach to describe and generate scientific knowledge for diverse architectures. This work presents a methodology to generate HPC applications,

independently from the platform, by using Archi-MDE.

Come4ACloud [184] is a generic architecture inspired by MDE principles to manage cloud systems autonomously. The main contribution of this proposal is the Autonomic Manager, an independent-layer cloud manager that finds optimal configurations of the infrastructure using a constraint solver.

Ostberg et al. presented CACTOS, a platform to automate and optimise cloud infrastructures [138]. The proposal has three main goals: modelling application and infrastructure resources, simulate the previously modelled application and resources, and automatically optimise the application deployment and resource usage. For this, the CACTOS platform consists of three main components: CactoScale, CactoOpt and CactoSim. CactoScale is a set of tools and techniques to collect and analyse application behaviour. CactoOpt consists of mathematical models to optimise application-resource mappings. CactoSim is a simulation platform for several application workloads, which can emulate data-centres and validate optimisation models in simulation environments.

Guerreiro et al. presented an MDE approach to estimate the performance and cost of cloud systems [68]. The authors considered IaaS and PaaS infrastructure levels for modelling cloud services with different levels of abstraction. Moreover, the proposal is designed from a perspective independent of the cloud provider.

Silvano et al. proposed ANTAREX [168], an approach based on DSLs and Aspect-Oriented Programming concepts to improve the applications with non-functional properties, such as performance, energy consumption and Quality of Service. The proposal is designed to work at design and run-time stages. Moreover, it provides a DSL based on LARA to supply extra-functional features to the application, such as parallelisation, mapping and adaptability aspects [40].

Although these works focus on modelling and analysing cloud systems, the level of detail of the infrastructures is generally low. That is, several characteristics of the computational and communication resources are not taken into account, such as CPU cores, memory delay, and network error ratio, which is considered as an important aspect to achieve the objectives of this thesis.

5.2 Modelling a cloud system

Designing cloud systems that provide an acceptable cost-performance ratio is challenging. Generally, a wide spectrum of components must be previously analysed, such as the kind of applications to be executed in the data-centre supporting the cloud, computing/storage requirements and the network topology, among others. Since each one of these components has a direct impact on the overall system performance, the design process is

complex and difficult, which usually requires the intervention of an expert. For this, we provide a framework for modelling, analysing and fixing misconfigurations of cloud systems. Section 5.2.1 describes a meta-model to properly represent the components of cloud systems, and Section 5.2.2 introduces a graphical concrete syntax to facilitate the process of designing a cloud infrastructure.

5.2.1 Abstract syntax

Figure 5.1 shows a simplified version of the proposed meta-model for represent data-centres. The *Data-centre* meta-class is the root class, which contains the main elements of the data-centre, such as those relating to both computational and networking aspects.

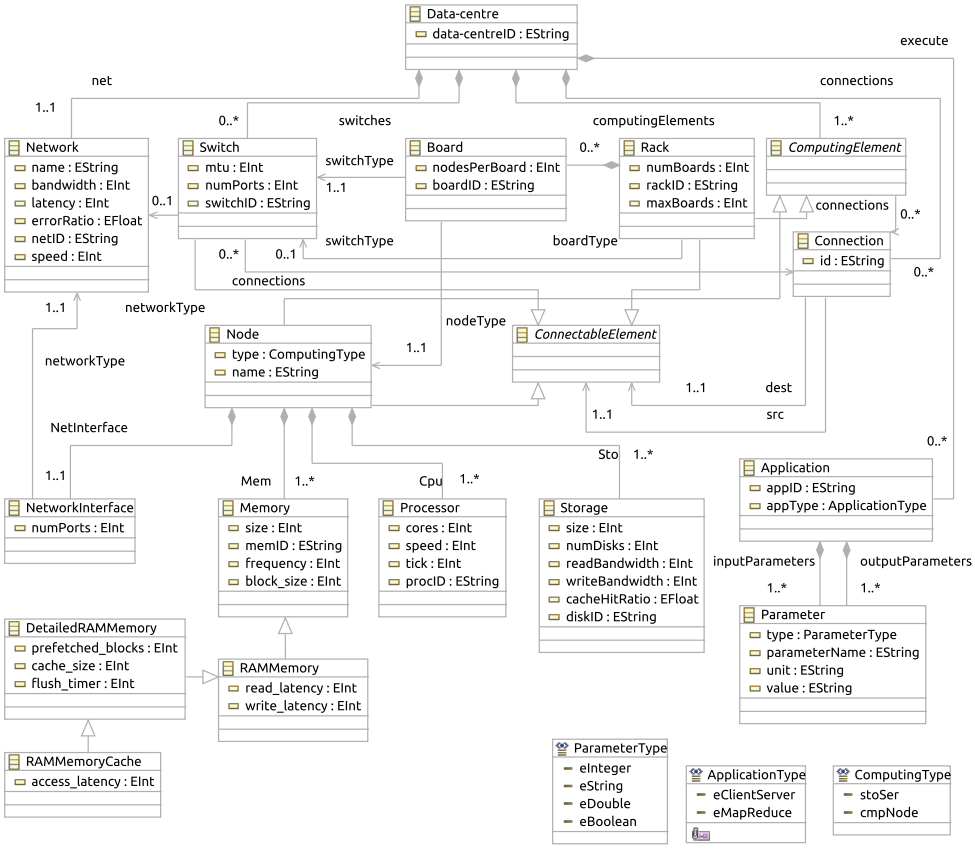


Figure 5.1: The data-centre meta-model (excerpt).

The computing elements can be divided in two categories. The first category of computing elements corresponds to the *Rack* meta-class. A rack represents a structure that contains multiple computing elements. In this case, it consists of a set of *Boards*, where each board includes several nodes

that are determined by the attribute *nodesPerBoard* and a switch that interconnect them. The second category corresponds to the *Node* meta-class, which represents a single node. A node is categorised by the attribute *type*, which denotes the purpose of the node. Computing nodes focus on providing high performance on computational operations while storage nodes are specialised in providing large storage with high transfer rates.

A node consist of four types of elements. The first one is the *Processor*, where *cores*, *speed* and *tick* represent the number of cores of each CPU, the CPU speed (measured in MIPS) and the period during which a process is allowed to run in a multitasking system without being forced to leave the CPU, respectively. The second element is the *Memory*, which can be categorised in 3 main classes with three common features, *size*, *frequency* and *blockSize*, which denote the total size of each module (measured in GBytes), the frequency of the memory (measured in Mhz) and the size of the memory blocks, respectively. The first class of memory is called *RAMMemory*, which represents a generic model of a RAM memory, where *readLatency* and *writeLatency* refer to the read and write memory latencies (measured in ns), respectively. The second class of memory is called *RAMMemoryCache*, which represents a RAM memory with cache features, where *cacheSize*, *flushTimer*, *prefetchedBlocks* and *accessLatency* represent the size of the cache memory (measured in KBytes), the period of time to induce the memory dump, the maximum number of blocks allowed in the prefetch operations and the memory latency (measured in ns), respectively. The last class of memory, known as *DetailedRAMMemory*, is a combination of the previous memory classes, whose attributes *cacheSize*, *flushTimer* and *prefetchedBlocks* have been previously defined. The third element of a node is the *Storage*, where *numDisks*, *size*, *readBandwidth*, *writeBandwidth* and *cacheHitRatio* are the number of disks, the size of each disk (measured in GBytes), the read and write bandwidth of the storage system (measured in Gbps) and the the ratio between the number of cache hits and the number of lookups, respectively. The last element represents the *NetworkInterface* of the node, where *numPorts* is the number of ports of the interface.

The communication network is defined by two elements. The *Network* meta-class represents the communication network of the data-centre, where *bandwidth*, *latency*, and *errorRatio* define the data transfer rate (measured in Gbps), the latency (measured in μs) and the error ratio of the network, respectively. The *Switch* meta-class represents a resource used to communicate the different elements of the data-centre, classified as *ConnectableElements*, through the communication network using the *Connection* meta-class. The main attributes of the switch, *mtu* and *numPorts* are the maximum transmission unit (measured in bytes) and the number of ports of the switch, respectively.

The applications that are deployed and executed in the data-centre are

modelled with the *Application* meta-class. The application is categorised by the attribute *appType*, which represents the type of the application and consists of a set of input and output parameters denoted by the *Parameter* meta-class. The *Parameter* consists of 4 attributes, *type*, *parameterName*, *unit* and *value*, which represent the type, name, unit and value of each parameter. Finally, the *Repository* meta-class (omitted from the figure) represents the data-centre repository, which provides an extensive collection of networking and computational components to model, with a high level of detail, a complete data-centre.

Figure 5.2: Example of a data-centre model.

Figure 5.2 depicts an example of a data-centre model that conforms the proposed meta-model. This model is inspired by a real IBM data-centre configuration, which consists of 1 *IBM Flex* rack and 1 *IBM v7000 Storage* rack. These racks are interconnected using a 40/10 Gigabit Ethernet communication network and a *SAN42B-R extension switch*. The *IBM Flex* rack consists of 6 *Flex System Enterprise Chassis* boards, where each board contains 14 *IBM Flex System p460* computing nodes. These boards consist of 4 CPUs with 8 cores, reaching a speed of 317.900 MIPS. The memory system consists of 32 slots, which contain 64 GB of RAM. Finally, the storage consists of two disks of 2 TBytes. The *IBM Storage* rack consists of 6 *Flex System Chassis Storage* boards, where each board contains 14 *IBM v7000* storage nodes. Each node consists of 2 CPUs with 4 cores, reaching an speed of 200.000 MIPS, 16 GB of RAM and 16 hard disks with a total storage of 10 TBytes.

5.2.2 Graphical concrete syntax

In addition to the tree-based syntax depicted in Figure 5.2, we have designed a graphical concrete syntax for the meta-model of Figure 5.1. To illustrate this syntax, Figure 5.3 shows the most common network topologies. From left to right and from top to bottom the figures depict the star (a), mesh (b), ring (c), tree (d), fully connected (e) and line(f) topologies.

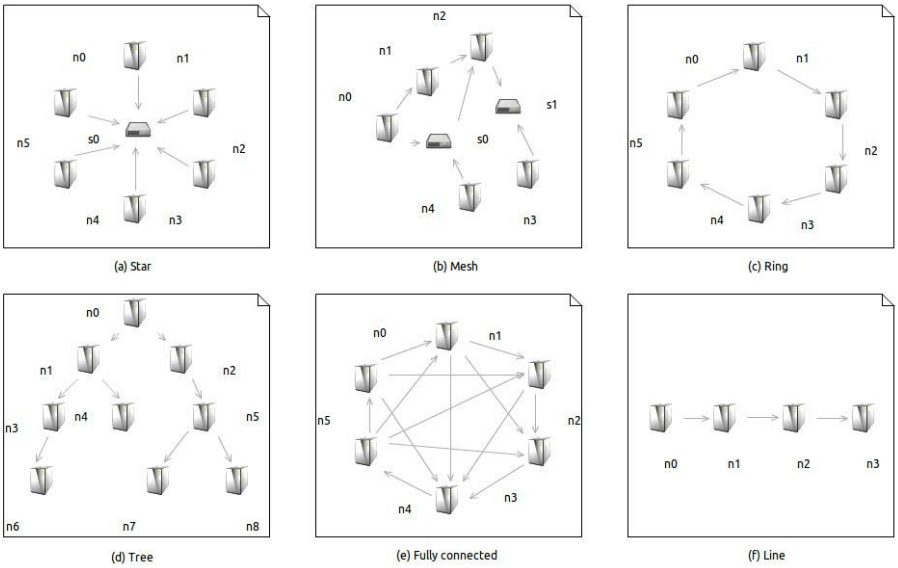


Figure 5.3: Well-known network topologies designed with the proposed graphical concrete syntax.

5.3 Expert rules

In order to support the user during the data-centre design process, this thesis includes a library of optimisation rules based on the knowledge of experts in designing data-centres. These expert rules aid the user to solve possible design issues, which in most cases, hamper the overall data-centre performance. However, expert rules must be designed and provided by an expert user, who must decide whether these are suitable to cover the requirements of the systems under study. The library consists of several rules aimed at analysing different features of the data-centre components, such as CPU processors, the memory system, storage and connectivity, among others. The main goal of these rules is to find inconsistencies in data-centre models and to provide relevant information to fix them. For the sake of simplicity, in this document we describe three sample rules from the complete library, which consists of up to 30 rules.

The expert rules are encoded in Epsilon Validation Language (in short, EVL) [102]. This language is similar to OCL, with the addition that it allows defining error levels, custom error messages and quick-fixes. Listing 6 shows the expert rule *CoresVsStorageNodesRatio*. This rule analyses the ratio between the number of storage nodes and the number of CPUs of a data-centre. The main objective of this rule is to avoid system bottlenecks caused by a reduced ratio of computational nodes between storage nodes. In this case, if the available storage nodes are not able to provide the required

```
1  context DataCentre
2  {
3    critique CoresVsStorageNodesRatio
4    {
5      check{
6        var storageNodes: Integer;
7        var totalCores: Integer;
8        storageNodes = self.calculateStorageNodes();
9        totalCores = self.calculateTotalCores();
10       return storageNodes*40 >= totalCores;
11     }
12     message: 'The number of storage nodes must be increased, there exist a high number of
           cores in comparison with the number of storage node which can act as bottleneck'
13   }
14 }
```

Listing 6: Data-centre topology optimisation rule encoded in EVL

performance, a message to modify the current design is shown. As it can be seen, the rule is applied on the context of DataCentre objects (line 1 of the listing). It is made of a check section (lines 5–11), which evaluates a certain condition on the model, and a message part, which is presented to the designer if the check part returns true.

Listing 7 shows two expert rules based on the analysis of two network features, bandwidth and latency. In this case, these rules check that these features range within a given interval. If some of these features is out of the range, the system provides a quick-fix to solve the issue. Quick-fixes are specified in the fix section of the rules (lines 8–11 and 18–21).

```
1  context Network
2  {
3    critique NetBandwidth
4    {
5      check: self.bandwidth>=10 and self.bandwidth <=100
6      message: 'Network '+ self.Name + ': bandwidth is usually ranged in [10−100].
7      Some of the most used configuration is 40'
8      fix {
9        title : "Set Bandwidth " + self.name + " bandwidth to 40"
10       do { self.bandwidth = 40; }
11     }
12   }
13   critique NetLatency
14   {
15     check: self.latency>=20 and self.latency <=2000
16     message: 'Network '+ self.name + ': latency is usually ranged in [20−2000].
17     Some of the most used configuration is 200'
18     fix {
19       title : "Set Latency " + self.name + " latency to 40"
20       do { self.latency = 200;}
21     }
22   }
23 }
```

Listing 7: Network optimisation rules encoded in EVL

5.4 Tool support

For designing the concrete syntax of the meta-model proposed in the previous section, and to implement the expert rules, we have used the Epsilon languages [103]. In addition to define the graphical concrete syntax of the meta-model, this set of languages allows performing several operations with models such as transformation, validation, comparison, migration and code generation, among others.

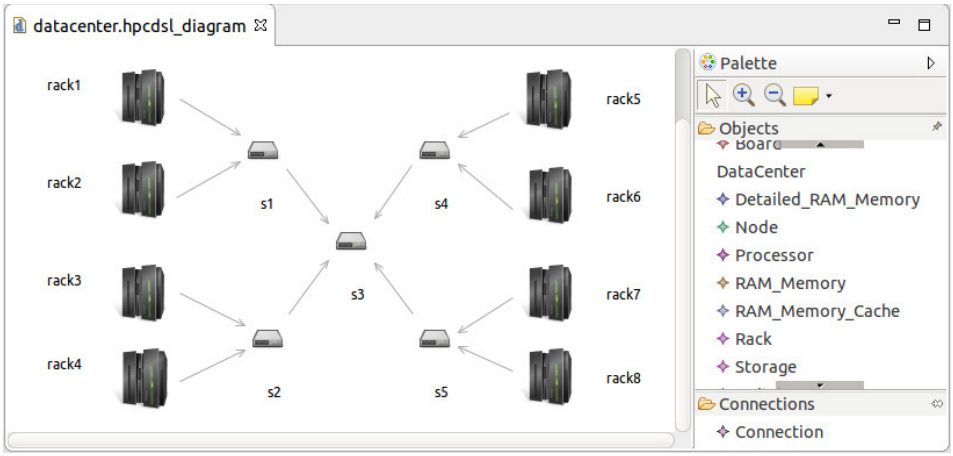


Figure 5.4: Editor of the graphical language.

The implementation and integration process consist of four steps. In the first step, the cloud meta-model has been created using Emfatic [54], a textual representation to create Ecore meta-models. In the second step, the meta-model is annotated using Epsilon's EuGENia, a tool for alleviating the complexity of Graphical Modeling Framework (in short, GMF) and Eclipse Modelling Framework (in short, EMF) [121]. In the third step, the graphical editor is created using EuGEnia [101] and several adjustments to the graphical cloud components are performed. Figure 5.4 illustrates the resulting graphical editor, which consists of three main components, a panel where the topology of the cloud infrastructure is represented by graphical items, a pallete that contains buttons to create the cloud components, and a property tab to show the different properties of the selected entities. The expert rules have been implemented using the EVL language, which allows to execute quick-fixes over the diagram as shown in the Figure 5.5.

In order to assess the performance and scalability of the designed data-centre models, they can be simulated, for which we use a code generator into the SIMCAN tool. An experimental phase to analyse the suitability of the framework, can be found in the paper associated with this contribution [36] (7.6).

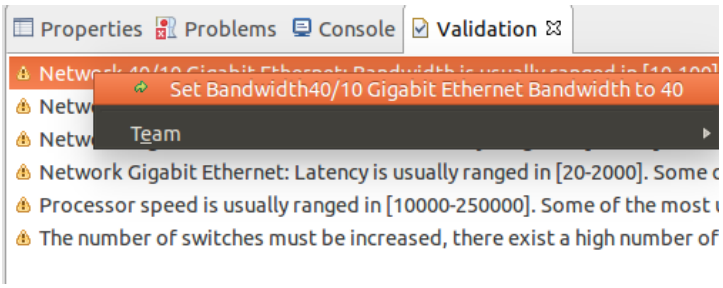


Figure 5.5: Quick fix.

5.5 Summary and conclusions

This chapter has presented the contributions made in the field of MDE applied to model and analyse cloud systems. For this, we provide a metamodel that properly represents the components of these systems, a set of expert rules for detecting and fixing design issues and a graphical language for easily designing cloud systems.

With these contributions it is possible to model and statically analyse cloud systems using structural rules. Moreover, the generated models can be dynamically analysed and optimised using the methodologies proposed in the previous chapters of this thesis.

5.6 Associated publications

(7.6) **MAGICIAN: Model-based design for optimizing the configuration of data-centres.**
Pablo C. Cañizares, Alberto Núñez and Juan de Lara.
In Proceedings of 29^{th} International Conference on Software Engineering and Knowledge Engineering, pp. 602-607, 2017.

Chapter 6

Conclusions and future work

Don't fall to sleep
Other people close their eyes
Seeing dreams in their sleep
But don't fall asleep

Tupac Shakur

This thesis has contributed with fundamental advances related to testing and modelling techniques for checking the correctness of cloud systems. Section 6.1 provides the conclusions of this thesis summarising the solutions and results achieved with each contribution of this work. Section 6.2 analyses and presents some future research lines to improve these contributions.

6.1 Conclusions

Concerning the field of MeT, this thesis proposes two contributions. The first one deals with a methodology that combines EAs and MeT to check the correctness of energy-aware cloud systems. This methodology is based on checking the satisfiability of MRs while testing cloud systems. The experimental results obtained from this study are promising, demonstrating that it is feasible to combine EAs and MeT to formally test cloud computing systems. This approach can not only be used to analyse the correctness of simulation platforms, but to discover flaws in cloud designs and to provide feasible solutions that improve these designs. The second contribution is a methodology for detecting faults in memory systems using simulation and MeT. For this purpose, a knowledge base using MRs was built, which addresses critical aspects of memory systems, such as functional, performance and energy consumption. To measure the effectiveness of our proposed methodology, an experimental study based on MuT was performed. In general, the proposed methodology achieves promising results, detecting the major part of the injected faults.

The thesis makes two contributions related to MuT. The first one is a framework for detecting errors in distributed applications executed in simulated environments. The obtained results show that the proposal is able to identify common mistakes made by competent programmers. A direct relation between the complexity of the application under test and the difficulty to detect errors was discovered. Hence, in simple applications, the major part of the mutants are detected by the entire test suite. Nevertheless, in complex applications, these errors are only detected by a reduced number of test cases. The second contribution is a set of optimisations to reduce the overall time required to execute this testing technique. The results show that the optimisations outperform previous proposals to improve the performance of the MuT process. In general, it provides the best results when different strategies are combined, obtaining in some scenarios an improvement of 70% in the overall performance with respect to other approaches found in the literature using HPC techniques.

Regarding MDE techniques, the thesis proposes a model-based approach for designing and analysing the data-centres supporting cloud systems. The methodology relies on expert rules to detect and fix suboptimal decisions and performs simulations to analyse the performance and scalability of the configurations. Several experiments were performed, modelling a real data-centre using the approach. In those experiments, the existent inconsistencies in the initial data-centre design were fixed after applying the suggestions proposed by the system. Moreover, the new designs provide an overall system performance higher than the initial model.

6.2 Future work

As future work, new advances in the three main lines of this thesis will be provided: MeT, MuT and MDE.

In the research line related to MeT, the testing methodology will be extended. Thus, users will be able to model both the software and hardware parts of cloud systems, design new cloud system models and automatically test these models using a cost-effective approach that considers both functional and non-functional aspects of the cloud. Moreover, the collection of MRs will be extended, providing a structured proposal to use diverse relations and interpret the obtained results. A DSL for specifying in a simple way the MRs to check the correctness of the system will be defined. Regarding the EAs, the trade-off between cost and energy consumption will be investigated. For this, a new EA dealing with the monetary cost of each component (e.g. CPUs, memories, networks) will be designed to provide relevant information to the user regarding the investment. That is, how the new hardware impacts on the overall energy efficiency. Finally, the integration of dynamic workloads, generated at run-time, into our framework will

be studied. The main difficulty of this task lies in how these workloads are compared in the MRs.

Regarding MuT, several bottlenecks that hamper the performance of the testing process were identified, like the compilation phase and the distribution algorithm when analysing real-world applications. Specifically, several drawbacks related to an elevated compilation time and a high level of communication and network traffic between the master and the worker processes were identified. In order to alleviate these issues, the scalability of different optimisations for parallelising the MuT process using large scale systems will be evaluated. Therefore, two solutions will be created. The first one is the parallelisation of the compilation phase to reduce the overall time. The second one is an adaptive distribution mechanism that changes the size of the execution grain depending on the remaining workload.

Regarding the MDE techniques, a semi-automatic tuning configuration to reach a specific performance goal will be supported. Recurring architectural patterns, which can be expressed as configurable templates will be identified. For this, an expert system will be built to assist the user in creating a cloud design. The expert system will ask questions to suggest the template that suits more the user's needs. In addition, the meta-model of the cloud system will be improved increasing the level of detail of the cloud model. From the point of view of the expert rules, these will be extended to analyse different aspects of the cloud, such as volume, performance/suitability and cost. Finally, the graphical language will be used as a front end of the methodologies provided along this thesis.

Bibliography

[1] Unified Modeling Language. Web page at https://www.omg.org/spec/UML/About-UML/. Date of last access: 24th September, 2019.

[2] A. Van-Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[3] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6):957–976, 2009.

[4] A. Ahmed and A. Sabyasachi. Cloud computing simulators: A detailed survey and future direction. In *IEEE International Advance Computing Conference*, IACC'14, pages 866–872, 2014.

[5] Amazon. Amazon Elastic Compute Cloud. Web page at http://aws.amazon.com/ec2/. Date of last access: 20th March, 2019.

[6] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2nd edition, 2017.

[7] J. Andrews, L. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *27th International Conference on Software Engineering*, ICSE'05, pages 402–411. ACM, 2005.

[8] V. Andrikopoulos, A. Reuter, S. Sáez, and F. Leymann. A GENTL approach for cloud application topologies. In *European Conference on Service-Oriented and Cloud Computing*, pages 148–159. Springer, 2014.

[9] A.R. da-Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.

[10] D. Arora and V. Bassi. *Generating Test Cases Using Metamorphic Testing and Genetic Algorithm for Integer Bugs Detection*. PhD thesis, 2015.

[11] M. Bahrami, O. Bozorg-Haddad, and X. Chu. Cat swarm optimization (CSO) algorithm. In *Advanced Optimization by Nature-Inspired Algorithms*, pages 9–18. Springer, 2018.

[12] R. Balasubramonian. Memory scheduling championship results. Accessed October, 2019.

[13] W. Banzhaf, L. Spector, and L. Sheneman. *Genetic Programming Theory and Practice XVI*. Springer, 2019.

[14] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[15] T. Benson, A. Akella, A. Shaikh, and S. Sahu. CloudNaaS: A cloud networking platform for enterprise applications. In *2nd ACM Symposium on Cloud Computing*, SOCC '11, page 8. ACM, 2011.

[16] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, and F. Leymann. A systematic review of cloud modeling languages. *ACM Computing Surveys*, 51(1):22, 2018.

[17] A. Bergmayr, A. Rossini, N. Ferry, G. Horn, L. Orue-Echevarria, A. Solberg, and M. Wimmer. The evolution of CloudML and its manifestations. In *3rd International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE)*, CloudMDE'15, pages 1–6, 2015.

[18] A. Bernal, M. Cambronero, V. Valero, A. Núñez, and P. Cañizares. A Framework for Modeling Cloud Infrastructures and User Interactions. *IEEE Access*, 7:43269–43285, 2019.

[19] C. Bienia, S. Kumar, J. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th International Conference on Parallel Architectures and Compilation Techniques*, PACT'08, pages 72–81. ACM, 2008.

[20] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[21] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann. TOSCA: Portable automated deployment and management of cloud applications. In *Advanced Web Services*, pages 527–549. Springer, 2014.

[22] G. Blair, N. Bencomo, and R. France. Models@ run. time. *Computer*, 42(10):22–27, 2009.

[23] M. Brambilla, J. Cabot, and M. Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[24] T. Budd and D. Angluin. Two Notions of Correctness and Their Relation to Testing. *Acta Informatica*, 18(1):31–45, 1982.

[25] J. Byrne, S. Svorobej, K. Giannoutakis, D. Tzovaras, P. Byrne, P. Östberg, A. Gourinovitch, and T. Lynn. A Review of Cloud Computing Simulation Platforms and Related Environments. In *7th International Conference on Cloud Computing and Services Science*, CLOSER'17, pages 651–663, 2017.

[26] C.-ai Sun, G. Wang, B. Mu, H. Liu, Z. Wang, and T. Chen. Metamorphic testing for web services: Framework and a case study. In *9th IEEE International Conference on Web Services*, ICWS'11, pages 283–290. IEEE, 2011.

[27] C.-ai Sun, G. Wang, B. Mu, H. Liu, Z. Wang, and T. Chen. A metamorphic relation-based approach to testing web services without oracles. *International Journal of Web Services Research*, 9(1):51–73, 2012.

[28] C.-ai Sun, G. Wang, Q. Wen, D. Towey, and T. Chen. MT4WS: An automated metamorphic testing system for web services. *International Journal of High Performance Computing and Networking*, 9(1/2):104–115, 2016.

[29] R. Calheiros, R. Ranjan, A. Beloglazov, C. Rose, and R. Buyya. CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. *Software - Practice and Experience*, 41(1):23–50, 2011.

[30] P. Cañizares, J. A. Núñez, and L. Llana. MT-EA4Cloud: A Methodology for testing and optimising energy-aware cloud systems. *Journal of Systems and Software*, –, 2019.

[31] P. Cañizares, M. Merayo, and A. Núñez. EMINENT: EMbarrassINgly parallEl mutatioN Testing. In *16th International Conference on Computational Science*, ICCS'16, pages 63–73, 2016.

[32] P. Cañizares, M. Merayo, and A. Núñez. FARTHEST: FormAl distRibuTed scHema to dEtect Suspicious arTefacts. In *8th Asian Conference on Intelligent Information and Database Systems*, ACIIDS'16, pages 770–779. Springer, 2016.

[33] P. Cañizares, M. Merayo, and A. Núñez. Using Ants to Fight Wildfire. In *14th International Work-Conference on Artificial Neural Networks*, IWANN'17, pages 371–380, Cadiz, Spain, 2017.

[34] P. Cañizares, M. Merayo, and A. Núñez. FORTIFIER: A FORmal disTrIbuted Framework to Improve the dEtection of thReatening objects in baggage. *Journal of Information Telecommunication*, 2(1):2–18, 2018.

[35] P. Cañizares, M. Merayo, and J. Vara. LAnt: Model driven approach for ant colony optimization. *Journal of Intelligent & Fuzzy Systems*, 32(2):1343–1354, 2017.

[36] P. Cañizares, A. Núñez, and J. de Lara. MAGICIAN: Model-based design for optimizing the configuration of data-centers. In *29th International Conference on Software Engineering and Knowledge Engineering*, SEKE'17, pages 602–607, Pittsburgh, PA, USA, 2017.

[37] P. Cañizares, A. Núñez, and J. de Lara. OUTRIDER: Optimizing the mUtation Testing pRocess In Distributed EnviRonments. In *17th International Conference on Computational Science*, ICCS'17, pages 505–514, Zurich, Switzerland, 2017.

[38] P. Cañizares, A. Núñez, and J. de Lara. An expert system for checking the correctness of memory systems using simulation and metamorphic testing. *Expert Systems with Applications*, 132:44–62, 2019.

[39] P. Cañizares, A. Núñez, and M. Merayo. Mutomvo: Mutation testing framework for simulated cloud and HPC environments. *Journal of Systems and Software*, 143:187–207, 2018.

[40] J. Cardoso, T. Carvalho, J. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov. LARA: An aspect-oriented programming language for embedded systems. In *11th Annual International Conference on Aspect-Oriented Software Development*, AOSD'12, pages 179–190. ACM, 2012.

[41] H. Casanova, A. Legrand, and M. Quinson. SimGrid: A Generic Framework for Large-Scale Distributed Experiments. In *10th International Conference on Computer Modeling and Simulation*, UKSIM' 08, pages 126–131, 2008.

[42] G. Castañé, A. Nunez, P. Llopis, and J. Carretero. E-mc2: A formal framework for energy modelling in cloud computing. *Simulation Modelling Practice and Theory*, 39:56–75, 2013.

[43] A. Cavalli, T. Higashino, and M. Núñez. A survey on formal active and passive testing with applications to the cloud. *Annales of Telecommunications*, 70(3-4):85–93, 2015.

[44] W. Chan, T. Chen, S. Cheung, T. Tse, and Z. Zhang. Towards the testing of power-aware software applications for wireless sensor networks.

In *12th International Conference on Reliable Software Technologies*, pages 84–99. Springer, 2007.

[45] W. Chan, T. Chen, H. Lu, T. Tse, and S. Yau. Integration testing of context-sensitive middleware-based applications: A metamorphic approach. *International Journal of Software Engineering and Knowledge Engineering*, 16(05):677–703, 2006.

[46] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti. USIMM: The Utah SImulated Memory Module A Simulation Infrastructure for the JWAC Memory Scheduling Championship. 2012.

[47] J. Chen, J. Jiang, and D. Luo. A predictive and evolutionary approach for cost-effective and deadline-constrained workflow scheduling over distributed IaaS clouds. *International Journal of Web Services Research*, 16(3):78–94, 2019.

[48] T. Chen, S. Cheung, and S. Yiu. Metamorphic testing: A new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.

[49] T. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys*, 51(1):4, 2018.

[50] T. Chen, F.-C. Kuo, H. Liu, and S. Wang. Conformance testing of network simulators based on metamorphic testing technique. In *11th IFIP WG 6.1 International Conference*, FMOODS'09, pages 243–248. Springer, 2009.

[51] T. Chen, F.-C. Kuo, R. Merkel, and W. Tam. Testing an open source suite for open queuing network modelling using metamorphic testing technique. In *14th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS'09, pages 23–29. IEEE, 2009.

[52] B. Choi and A. Mathur. High-performance Mutation Testing. *Journal of Systems and Software*, 20(2):135–152, 1993.

[53] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer. On the number and nature of faults found by random testing. *Software Testing, Verification and Reliability*, 21(1):3–28, 2011.

[54] C. Daly, M. Garcia, and L.Bigeardel. Emfatic language for EMF development. 2004. Accessed October, 2019.

[55] M. Delamaro, J. Maldonado, and A. Mathur. Integration testing using interface mutation. In *7th International Symposium on Software Reliability Engineering*, ISSRE'96, pages 112–121, 1996.

[56] R. DeMillo, R. Lipton, and F. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, 1978.

[57] J. Ding, T. Wu, D. Wu, J. Lu, and X.-H. Hu. Metamorphic testing of a Monte Carlo modeling program. In *6th International Workshop on Automation of Software Test*, AST'11, pages 1–7. ACM, 2011.

[58] J. Ding, D. Zhang, and X.-H. Hu. An application of metamorphic testing for testing scientific software. In *1st International Workshop on Metamorphic Testing*, MET'16, pages 37–43. ACM, 2016.

[59] M. Dorigo and T. Stützle. Ant colony optimization: Overview and recent advances. In *Handbook of Metaheuristics*, pages 311–351. Springer, 2019.

[60] L. Drummond. A GPU-Based metaheuristic for workflow scheduling on clouds. In *13th International Conference High Performance Computing for Computational Science-Vecpar*, volume 11333 of *VECPAR'19*, page 62. Springer, 2019.

[61] E. Elbeltagi, T. Hegazy, and D. Grierson. Comparison among five evolutionary-based optimization algorithms. *Advanced engineering informatics*, 19(1):43–53, 2005.

[62] F.-C. Kuo, T. Chen, and W. Tam. Testing embedded software by metamorphic testing: A wireless metering system case study. In *36th Conference on Local Computer Networks*, LCN'11, pages 291–294. IEEE, 2011.

[63] F. Fakhfakh, H. Kacem, and A. Kacem. Simulation tools for cloud computing: A survey and comparative study. In *16th International Conference on Computer and Information Science*, ICIS'17, pages 221–226, 2017.

[64] N. Ferry, F. Chauvel, H. Song, A. Rossini, M. Lushpenko, and A. Solberg. Cloudmf: Model-driven management of multi-cloud applications. *ACM Transactions on Internet Technology*, 18(2):16, 2018.

[65] M. Filho, R. Oliveira, C. Monteiro, P. Inácio, and M. Freire. CloudSim Plus: A cloud computing simulation framework pursuing software engineering principles for improved modularity, extensibility and correctness. In *IFIP/IEEE Symposium on Integrated Network and Service Management*, IM'17, pages 400–406, 2017.

[66] S. Frey and W. Hasselbring. The cloudmig approach: Model-based migration of software systems to cloud-optimized applications. *International Journal on Advances in Software*, 4(3 and 4):342–353, 2011.

[67] W. Grosso. *Java RMI*. O'Reilly & Associates, Inc., 1st edition, 2001.

[68] M. Guerriero, M. Ciavotta, G. Gibilisco, and D. Ardagna. SPACE4Cloud: A DevOps environment for multi-cloud applications. In *1st International Workshop on Quality-Aware DevOps*, QUDOS'15, pages 29–30. ACM, 2015.

[69] J. Guillén, J. Miranda, J. Murillo, and C. Canal. A UML Profile for modeling multicloud applications. In *2nd European Conference on Service-Oriented and Cloud Computing*, ESOCC'13, pages 180–187. Springer, 2013.

[70] M. Hamdaqa and L. Tahvildari. Stratus ML: A layered cloud modeling framework. In *{{IEEE International Conference}} on {{Cloud Engineering}}*, IC2E'15, pages 96–105. IEEE, 2015.

[71] R. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, 1977.

[72] D. Harel and B. Rumpe. Meaningful modeling: What's the semantics of" semantics"? *Computer*, 37(10):64–72, 2004.

[73] M. Harman, Y. Jia, and Y. Zhang. Achievements, Open Problems and Challenges for Search Based Software Testing. In *IEEE 8th International Conference on Software Testing, Verification and Validation*, ICST'15, pages 1–12, 2015.

[74] F. Hermans, M. Pinzger, and A. Van-Deursen. *Domain-Specific Languages in Practice: A User Study on the Success Factors*. Springer, 2009.

[75] R. Hierons, K. Bogdanov, J. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A. Simons, S. Vilkomir, M. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2):9:1–9:76, 2009.

[76] R. Hierons, M. Harman, and S. Danicic. Using Program Slicing to Assist in the Detection of Equivalent Mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[77] T. Holmes. Facilitating Development and Provisioning of Service Topologies through Domain-Specific Languages. In *18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations*, pages 422–425. IEEE, 2014.

[78] S. Hoops, S. Sahle, R. Gauges, C. Lee, J. Pahle, N. Simus, M. Singhal, L. Xu, P. Mendes, and U. Kummer. COPASI—a complex pathway simulator. *Bioinformatics*, 22(24):3067–3074, 2006.

[79] F. Howell and R. Mcnab. Simjava: A Discrete Event Simulation Library For Java. In *International Conference on Web-Based Modeling and Simulation*, pages 51–56, 1998.

[80] S. Hummel, E. Schonberg, and L. Flynn. Factoring: A Method for Scheduling Parallel Loops. *Journal of Communications of ACM*, 35(8):90–101, 1992.

[81] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria. In *16th International Conference on Software Engineering*, ICSE'94, pages 191–200, 1994.

[82] J. Hutchinson, M. Rouncefield, and J. Whittle. Model-driven engineering practices in industry. In *33rd International Conference on Software Engineering*, ICSE'11, pages 633–642. ACM, 2011.

[83] J. Hutchinson, J. Whittle, and M. Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, 89:144–161, 2014.

[84] H. Ibrahim, R. Aburukba, and K. El-Fakih. An Integer Linear Programming model and Adaptive Genetic Algorithm approach to minimize energy consumption of Cloud computing data centers. *Computers & Electrical Engineering*, 67:551–565, 2018.

[85] G. Ismayilov and H. Topcuoglu. Neural network based multi-objective evolutionary algorithm for dynamic workflow scheduling in cloud computing. *Future Generation Computer Systems*, 102:307–322, 2020.

[86] K. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. Wasserman, and N. Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *International Conference on Cloud Computing Technology and Science*, pages 159–168. IEEE, 2010.

[87] C. Jatoth, G. Gangadharan, and R. Buyya. Optimal fitness aware cloud service composition using an adaptive genotypes evolution based genetic algorithm. *Future Generation Computer Systems*, 94:185–198, 2019.

[88] M. Jeong, D. Yoon, and M. Erez. DrSim: A platform for flexible DRAM system research. 2012.

[89] Y. Jia and M. Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Testing: Academic & Industrial Conference - Practice and Research Techniques*, TAIC'08, pages 94–98. IEEE, 2008.

[90] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[91] M. Jiang, T. Chen, F.-C. Kuo, and Z. Ding. Testing central processing unit scheduling algorithms using metamorphic testing. In *4th International Conference on Software Engineering and Service Science*, ICSESS'13, pages 530–536. IEEE, 2013.

[92] K. Jong. *Evolutionary Computation: A Unified Approach*. A Bradford Book, march 25, 2016 edition, 2016.

[93] A. JoSEP, R. KAtz, A. KonWinSKi, L. Gunho, D. PAttERSon, and A. RABKin. A view of cloud computing. *Communications of the ACM*, 53(4), 2010.

[94] V. Kachitvichyanukul. Comparison of three evolutionary algorithms: GA, PSO, and DE. *Industrial Engineering and Management Systems*, 11(3):215–223, 2012.

[95] G. Kecskemeti. DISSECT-CF: A simulator to foster energy-aware scheduling in infrastructure clouds. *Simulation Modelling Practice and Theory*, 58:188–218, 2015. Special issue on Cloud Simulation.

[96] G. Keller, M. Tighe, H. Lutfiyya, and M. Bauer. DCSim: A data centre simulation tool. In *2013 IFIP/IEEE International Symposium on Integrated Network Management*, IM'13, pages 1090–1091. IEEE, 2013.

[97] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, 2008.

[98] B. Keshanchi, A. Souri, and N. Navimipour. An improved genetic algorithm for task scheduling in the cloud environments using the priority queues: Formal verification, simulation, and statistical testing. *Journal of Systems and Software*, 124:1–21, 2017.

[99] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, 2016.

[100] W. Kolberg, P. Marcos, J. Anjos, A. Miyazaki, C. Geyer, and L. Arantes. Mrsg–a mapreduce simulator over simGrid. *Parallel Computing*, 39(4-5):233–244, 2013.

[101] D. Kolovos, A. García-Domínguez, L. Rose, and R. Paige. Eugenia: Towards disciplined and automated development of GMF-based graphical model editors. *Software & Systems Modeling*, 16(1):229–255, 2017.

[102] D. Kolovos, R. Paige, and F. Polack. Rigorous methods for software construction and analysis. chapter On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages, pages 204–218. Springer-Verlag, Berlin, Heidelberg, 2009.

[103] D. Kolovos, L. Rose, A. García-Domínguez, and R. Paige. The epsilon book. 2010.

[104] E. Krauser, A. Mathur, and V. Rego. High Performance Software Testing on SIMD Machines. *IEEE Transactions on Software Engineering*, 17(5):403–423, 1991.

[105] K. Kritikos, J. Domaschka, and A. Rossini. SRL: A scalability rule language for multi-cloud environments. In *6th International Conference on Cloud Computing Technology and Science*, CLOSER'16, pages 1–9. IEEE, 2014.

[106] T. Krüger, T. Davidović, D. Teodorović, and M. Šelmić. The bee colony optimization algorithm and its convergence. *International Journal of Bio-Inspired Computation*, 8(5):340–354, 2016.

[107] M. Kusano and C. Wang. CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In *28th International Conference on Automated Software Engineering*, ASE'13, pages 722–725, 2013.

[108] L. Lamport. *LATEX: A Document Preparation System.* Addison-Wesley, 2nd edition, 1994.

[109] V. Le, M. Afshari, and Z. Su. Compiler Validation via Equivalence Modulo Inputs. In *35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'14, pages 216–226. ACM, 2014.

[110] J. Li, M. Humphrey, C. van Ingen, D. Agarwal, K. Jackson, and Y. Ryu. eScience in the cloud: A MODIS satellite data reprojection and reduction pipeline in the Windows Azure platform. In *International Symposium on Parallel Distributed Processing*, IPDPS'10, pages 1–10, 2010.

[111] R. Lipton and F. Sayward. The Status of Research on Program Mutation. In *Workshop on Software Testing and Test Documentation*, pages 355–373, 1978.

[112] H. Liu, F.-C. Kuo, D. Towey, and T. Chen. How Effectively Does Metamorphic Testing Alleviate the Oracle Problem? *IEEE Transactions on Software Engineering*, 40(1):4–22, 2014.

[113] Y. Ma and S. Kim. Mutation testing cost reduction by clustering overlapped mutants. *Journal of Systems and Software*, 115:18–30, 2016.

[114] Y. Ma, A. Offutt, and Y. Kwon. MuJava: An Automated Class Mutation System: Research Articles. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[115] Y. Mansouri, A. Toosi, and R. Buyya. Data Storage Management in Cloud Environments: Taxonomy, Survey, and Future Directions. *ACM Computing Surveys*, 50(6):91:1–91:51, 2017.

[116] A. Mathur and E. Krauser. Modeling mutation and a vector processor. In *10th International Conference on Software Engineering*, ICSE'88, pages 154–161, 1988.

[117] I. MathWorks. *MATLAB: The Language of Technical Computing. Desktop Tools and Development Environment*, volume 9. MathWorks, 7nd edition, 2005.

[118] L. Mei, W. Chan, and T. Tse. A Tale of Clouds: Paradigm Comparisons and Some Thoughts on Research Issues. In *3rd IEEE Asia-Pacific Services Computing Conference*, APSCC'08, pages 464–469. IEEE Computer Society, 2008.

[119] P. Mell and T. Grance. SP 800-145. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards & Technology, U.S. Department of Commerce, Gaithersburg, MD, United States, 2011.

[120] S. Mellor, S. Kendall, A. Uhl, and D. Weise. *MDA Distilled*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[121] E. Merks, R. Eliersick, T. Grose, F. Budinsky, and D. Steinberg. The eclipse modeling framework. *retrieved from, total*, page 37, 2003.

[122] S. Mirjalili. Genetic algorithm. In *Evolutionary Algorithms and Neural Networks*, pages 43–55. Springer, 2019.

[123] S. Mirjalili. Particle swarm optimisation. In *Evolutionary Algorithms and Neural Networks*, pages 15–31. Springer, 2019.

[124] C. Murphy, M. Raunak, A. King, S. Chen, C. Imbriano, G. Kaiser, I. Lee, O. Sokolsky, L. Clarke, and L. Osterweil. On Effective Testing of Health Care Simulation Software. In *3rd Workshop on Software Engineering in Health Care*, SEHC '11, pages 40–47, 2011.

[125] G. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. John Wiley & Sons, 3rd edition, 2011.

[126] A. Namin and J. Andrews. Finding Sufficient Mutation Operators via Variable Reduction. In *2nd International Workshop on Mutation Analysis*, MUTATION'06, pages 5–5, 2006.

[127] A. Núñez, J. Fernández, R. Filgueira, F. García, and J. Carretero. SIMCAN: A flexible, scalable and expandable simulation platform for modelling and simulating distributed architectures and applications. *Simulation Modelling Practice and Theory*, 20(1):12–32, 2012.

[128] A. Núñez and R. Hierons. A methodology for validating cloud models using metamorphic testing. *Annales of Telecommunications*, 70(3-4):127–135, 2015.

[129] A. Núñez, J.L. Vázquez-Poletti, A. Caminero, G. Castañé, J. Carretero, and I. Llorente. iCanCloud: A Flexible and Scalable Cloud Infrastructure Simulator. *Journal of Grid Computing*, 10(1):185–209, 2012.

[130] A. Offutt. The Coupling Effect: Fact or Fiction. *ACM SIGSOFT Software Engineering Notes*, 14(8):131–140, 1989.

[131] A. Offutt and W. Craft. Using Compiler Optimization Techniques to Detect Equivalent Mutants. *Software Testing, Verification and Reliability*, 4(3):131–154, 1994.

[132] A. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997.

[133] A. Offutt, R. Pargas, S. Fichter, and P. Khambekar. Mutation Testing of Software Using a MIMD Computer. In *21st International Conference on Parallel Processing*, ICPP'92, pages 255–266, 1992.

[134] M. Olsen and M. Raunak. Increasing Validity of Simulation Models Through Metamorphic Testing. *IEEE Transactions on Reliability*, (99):1–18, 2018.

[135] H. Ouarnoughi, J. Boukhobza, F. Singhoff, and S. Rubini. Integrating I/Os in Cloudsim for Performance and Energy Estimation. *ACM SIGOPS Operating Systems Review*, 50(1):27–36, 2017.

[136] P. Delgado-Pérez, I. Medina-Bulo, J.J. Domínguez-Jiménez, A. García-Domínguez, and F. Palomo-Lozano. Class mutation operators for C++ object-oriented systems. *Annales of Telecommunications*, 70(3):137–148, 2014.

[137] P. Delgado-Pérez, S. Segura, and I. Medina-Bulo. Assessment of C++ object-oriented mutation operators: A selective mutation approach. *Software Testing, Verification and Reliability*, 2017.

[138] P.-O. Ostberg, H. Groenda, S. Wesner, J. Byrne, D. Nikolopoulos, C. Sheridan, J. Krzywda, A. Ali-Eldin, J. Tordsson, and E. Elmroth. The CACTOS vision of context-aware cloud topology optimization and simulation. In *6th International Conference on Cloud Computing Technology and Science*, CloudCom'14, pages 26–31. IEEE, 2014.

[139] R. Paige, D. Kolovos, and F. Polack. A tutorial on metamodelling for grammar researchers. *Science of Computer Programming*, 96:396–416, 2014.

[140] M. Palyart, D. Lugato, I. Ober, and J.-M. Bruel. MDE4HPC: An approach for using model-driven engineering in high-performance computing. In *15th International SDL Forum Toulouse*, pages 247–261. Springer, 2011.

[141] M. Papadakis, Y. Jia, M. Harman, and Y. Traon. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *7th International Conference on Software Engineering*, ICSE '15, pages 936–946. IEEE Press, 2015.

[142] K. Park and V. Pai. CoMon: A mostly-scalable monitoring system for PlanetLab. *ACM SIGOPS Operating Systems Review*, 40(1):65–74, 2006.

[143] D. Piatov, A. Janes, A. Sillitti, and G. Succi. Using the eclipse C/C++ development tooling as a robust, fully functional, actively maintained, open source c++ parser. In *8th IFIP WG 2.13 International Conference on Open Source Systems: Long-Term Sustainability*, volume 378 of *OSS'12*, page 399. Springer, 2012.

[144] C. Quinton, D. Romero, and L. Duchien. Cardinality-based feature models with constraints: A pragmatic approach. In *17th International Conference on Software Product Line*, pages 162–166. ACM, 2013.

[145] K. Qureshi and H. Rashid. A performance evaluation of rpc, java rmi, mpi and pvm. *Malaysian Journal of Computer Science*, 18(2):38–44, 2005.

[146] P. Rao, Z. Zheng, T. Chen, N. Wang, and K. Cai. Impacts of Test Suite's Class Imbalance on Spectrum-Based Fault Localization Techniques. In *13th International Conference on Quality Software*, QSIC'13, pages 260–267, 2013.

[147] P. Reales and M. Polo. Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases. In *28th International Conference on Software Maintenance*, ICSME'12, pages 646–649. IEEE, 2012.

[148] P. Reales and M. Polo. Parallel mutation testing. *Software Testing, Verification and Reliability*, 23(4):315–350, 2013.

[149] V. Rego and A. Mathur. Concurrency enhancement through program unification: A performance analysis. *Journal of Parallel and Distributed Computing*, 8(3):201–217, 1990.

[150] M. Richters and M. Gogolla. On formalizing the UML object constraint language OCL. In *International Conference on Conceptual Modeling*, pages 449–464. Springer, 1998.

[151] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[152] A. Rossini, K. Kritikos, N. Nikolov, J. Domaschka, F. Griesinger, D. Seybold, D. Romero, M. Orzechowski, G. Kapitsaki, and A. Achilleos. The cloud application modelling and execution language (CAMEL). 2017.

[153] J. Rounds and U. Kanewala. Systematic testing of genetic algorithms: A metamorphic testing based approach. *arXiv preprint arXiv:1808.01033*, 2018.

[154] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2004.

[155] M. Rutherford, A. Carzaniga, and A. Wolf. Simulation-Based Testing of Distributed Systems. Technical Report CU-CS-1004-06, Department of Computer Science, University of Colorado, 2006.

[156] I. Saleh and K. Nagi. HadoopMutator: A Cloud-Based Mutation Testing Framework. In *Software Reuse for Dynamic Systems in the Cloud and Beyond*, pages 172–187. Springer, 2014.

[157] J. Schad. Flying yellow elephant: Predictable and efficient MapReduce in the cloud. *Information Systems Group, Saarland University*, 2010.

[158] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *VLDB Endowment*, 3(1-2):460–471, 2010.

[159] D. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39:25– 31, 2006.

[160] D. Schuler and A. Zeller. Javalanche: Efficient Mutation Testing for Java. In *16th International Conference of Software Engineering Conference and the ACM SIGSOFT Symposium*, ESEC/FSE'09, pages 297–298. ACM, 2009.

[161] S. Segura, G. Fraser, A. Sanchez, and A. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016.

[162] S. Segura, J. Parejo, J. Troya, and A. Ruiz-Cortés. Metamorphic Testing of RESTful Web APIs. *IEEE Transactions on Software Engineering*, 2017.

[163] S. Segura, J. Troya, A. Durán, and A. Ruiz-Cortés. Performance Metamorphic Testing: A Proof of Concept. *Information and Software Technology*, 2018.

[164] E. Seidewitz. What Models Mean. *IEEE Software*, 20(5):26–32, 2003.

[165] R. Shannon. Introduction to the art and science of simulation. In *30th Conference on Winter Simulation*, WSC'98, pages 7–14. IEEE Computer Society Press, 1998.

[166] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE micro*, 23(6):84–93, 2003.

[167] G. Silva, L. Rose, and R. Calinescu. Cloud DSL: A Language for Supporting Cloud Portability by Describing Cloud Entities. In *2nd International Workshop on Model-Driven Engineering on and for the Cloud*, CloudMDE'14, pages 36–45, 2014.

[168] C. Silvano, G. Agosta, A. Bartolini, A. Beccari, L. Benini, L. Besnard, J. Bispo, R. Cmar, J. Cardoso, C. Cavazzoni, et al. The ANTAREX Domain Specific Language for High Performance Computing. *arXiv preprint arXiv:1901.06175*, 2019.

[169] Su Te Lei and Kang Zhang and Kei-Chun Li. Experience with the design of a performance tuning tool for parallel programs. *Journal of Systems and Software*, 39(1):27–37, 1997.

[170] P. B. S.U.K. Dzmitry Kliazovich. GreenCloud: A packet-level simulator of energy-aware cloud computing data centers. *The Journal of Supercomputing*, 62(3):1263–1283, 2012.

[171] T. Berners-Lee and D. Connolly. Hypertext markup language-2.0. *RFC 1866*, 1995.

[172] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *London Mathematical Society*, 2(42):230–265, 1936.

[173] R. Untch, A. Offutt, and M. Harrold. Mutation Analysis Using Mutant Schemata. In *3rd International Symposium on Software Testing and Analysis*, ISSTA'93, pages 139–148, 1993.

[174] M. Usaola and P. Mateo. Mutation testing cost reduction techniques: A survey. *IEEE Software*, 27(3):80–86, 2010.

[175] M. Vasudevan, Y.-C. Tian, M. Tang, E. Kozan, and X. Zhang. Energy-efficient application assignment in profile-based data center management through a Repairing Genetic Algorithm. *Applied Soft Computing*, 67:399–408, 2018.

[176] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specic Languages*. dslbook.org, 2013. URL http://www.dslbook.org, 2013.

[177] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Professional, 2003.

[178] J. Wegener, K. Grimm, M. Grochtmann, H. Sthamer, and B. Jones. Systematic testing of real-time systems. In *4th International Conference on Software Testing Analysis and Review (EuroSTAR 96)*, 1996.

[179] A. Weiss. Computing in the clouds. *netWorker*, 11(4):16–25, 2007.

[180] E. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[181] K. Wloch and P. Bentley. Optimising the performance of a formula one car using a genetic algorithm. In *International Conference on Parallel Problem Solving from Nature*, pages 702–711. Springer, 2004.

[182] Z. Xiao, J. Jiang, Y. Zhu, Z. Ming, S. Zhong, and S. Cai. A solution of dynamic VMs placement problem for energy consumption optimization based on evolutionary game theory. *Journal of Systems and Software*, 101(C):260–272, 2015.

[183] X. Xie, W. Wong, T. Chen, and B. Xu. Metamorphic slice: An application in spectrum-based fault localization. *Information and Software Technology*, 55(5):866–879, 2013.

[184] Z. Al-Shara, F. Alvares, H. Bruneliere, J. Lejeune, C. Prud'Homme, and T. Ledoux. Come4acloud: An end-to-end framework for autonomic cloud systems. *Future Generation Computer Systems*, 86:339–354, 2018.

[185] I. Zelinka. A survey on evolutionary algorithms dynamics and its complexity – Mutual relations, past, present and future. *Swarm and Evolutionary Computation*, 25:2 – 14, 2015.

[186] Z. Zhou, S. Zhang, M. Hagenbuchner, T. Tse, F.-C. Kuo, and T. Chen. Automated functional testing of online search services. *Software Testing, Verification and Reliability*, 22(4):221–243, 2012.

Part II

Papers Related to This Thesis

Chapter 7

List of publications included in this thesis

Journals:

(7.1) **An expert system for checking the correctness of memory systems using simulation and metamorphic testing.**
Pablo C. Cañizares, Alberto Núñez and Juan de Lara.
Expert Systems with Applications 132: pp. 44-62, 2019.
JCR Ranking: 20/132, (**Q1**), impact factor: 4.292, in **Computer Science, Artificial Intelligence**, DOI: 10.1016/j.eswa.2019.04.070

(7.3) **Mutomvo: Mutation testing framework for simulated cloud and HPC environments.**
Pablo C. Cañizares, Alberto Núñez and Mercedes G. Merayo.
Journal of Systems and Software 143: pp. 187-207, 2018.
JCR Ranking: 19/104, (**Q1**) impact factor: 2.559, in **Computer Science, Software Engineering**, DOI: 10.1016/j.jss.2018.05.010

(7.9) **LAnt: Model driven approach for ant colony optimization.**
Pablo C. Cañizares, Mercedes G. Merayo and Juan M. Vara.
Journal of Intelligent & Fuzzy Systems 32: pp. 1343-1354, 2017.
JCR Ranking: 76/132, (**Q3**) impact factor: 1.426, in **Computer Science, Artificial Intelligence**, DOI: 10.3233/JIFS-169132

(7.8) **FORTIFIER: A FORmal disTrIbuted Framework to Improve the dEtection of thReatening objects in baggage.**
Pablo C. Cañizares, Mercedes G. Merayo and Alberto Núñez.
Journal of Information Telecommunication 2: pp. 2-18, 2017.
DOI: 10.3233/JIFS-169132

Journals under review:

(7.2) **MT-EA4Cloud: A Methodology for testing and optimizing energy-aware cloud systems.**
Pablo C. Cañizares, Alberto Núñez, Juan de Lara and Luis Llana.
Journal of Systems and Software (under 3^{rd} round of revision)
JCR Ranking: 20/132, (**Q1**), impact factor: 4.292, in **Computer Science**, DOI: 10.1016/j.eswa.2019.04.070

International Conferences:

(7.4) **EMINENT: EMbarrassINgly parallEl mutatioN Testing.**
Pablo C. Cañizares, Mercedes G. Merayo and Alberto Núñez.
In Proceedings of 16^{th} International Conference on Computational Science. pp. 63-73, 2016.
CORE Ranking: A, DOI: 10.1016/j.procs.2016.05.298

(7.5) **OUTRIDER: Optimizing the mUtation Testing pRocess In Distributed EnviRonments.**
Pablo C. Cañizares, Alberto Núñez and Juan de Lara.
In Proceedings of 17^{th} International Conference on Computational Science. pp. 505-514, 2017.
CORE Ranking: A, DOI: 10.1016/j.procs.2017.05.095

(7.6) **MAGICIAN: Model-based design for optimizing the configuration of data-centres.**
Pablo C. Cañizares, Alberto Núñez and Juan de Lara.
In Proceedings of 29^{th} International Conference on Software Engineering and Knowledge Engineering, pp. 602-607, 2017.
CORE Ranking: B, DOI: doi.org/10.18293/SEKE2017-108

(7.10) **Using ants to fight wildfires.**
Pablo C. Cañizares, Alberto Núñez and Mercedes G. Merayo.
In Proceedings of 14th International Work-Conference on Artificial Neural Networks, pp. 371-380, 2017.
CORE Ranking: B, DOI: 10.1007/978-3-319-59147-6_32

(7.7) **FARTHEST: FormAl distRibuTed scHema to dEtect Suspicious arTefacts.**
Pablo C. Cañizares, Mercedes G. Merayo and Alberto Núñez.
In Proceedings of 8^{th} Asian Conference on Intelligent Information and Database Systems, pp. 770-779, 2016.
CORE Ranking: –, DOI: 10.1007/978-3-662-49381-6_74

7.1 An expert system for checking the correctness of memory systems using simulation and metamorphic testing

7.1	
Publication data	
Authors:	Pablo C. Cañizares, Alberto Núñez and Juan de Lara
Title:	An expert system for checking the correctness of memory systems using simulation and metamorphic testing
Publication:	Expert Systems with Applications
Pub. Type:	Journal
Year:	2019
DOI/URL:	https://doi.org/10.1016/j.eswa.2019.04.070
Pages:	19
Category:	Computer Science, Artificial Intelligence
Quartile:	Q1
Ranking:	24/133
Impact factor:	4.292
Contribution	
Summary:	This paper presents a methodology for checking the correctness of memory systems, which are considered an important subsystem for the overall performance of a cloud system. The methodology combines simulation and MeT to test memory systems. The novelty of the methodology lies in an expert system to properly analyse memory systems. The knowledge of the expert is represented in the form of metamorphic relations, which are properties of the analysed system involving multiple inputs and their outputs
Technique:	Metamorphic Testing
Secondary techniques:	Simulation, Formal Modelling, Mutation Testing, AI

Expert Systems With Applications 132 (2019) 44–62

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

An expert system for checking the correctness of memory systems using simulation and metamorphic testing

Pablo C. Cañizares[a,*], Alberto Núñez[a], Juan de Lara[b]

[a] Dept. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Spain
[b] Dept. Ingeniería Informática, Universidad Autónoma de Madrid, Spain

ARTICLE INFO

Article history:
Received 17 August 2018
Revised 5 April 2019
Accepted 30 April 2019
Available online 1 May 2019

Keywords:
Memory systems
Metamorphic testing
Simulation
Mutation testing
Expert systems
Memory scheduling

ABSTRACT

During the last few years, computer performance has reached a turning point where computing power is no longer the only important concern. This way, the emphasis is shifting from an exclusive focus on the optimisation of the computing system to optimising other systems, like the memory system. Broadly speaking, testing memory systems entails two main challenges: the oracle problem and the reliable test set problem. The former consists in deciding if the outputs of a test suite are correct. The latter refers to providing an appropriate test suite for determining the correctness of the system under test.

In this paper we propose an expert system for checking the correctness of memory systems. In order to face these challenges, our proposed system combines two orthogonal techniques – simulation and metamorphic testing – enabling the automatic generation of appropriate test cases and deciding if their outputs are correct. In contrast to conventional expert systems, our system includes a factual database containing the results of previous simulations, and a simulation platform for computing the behaviour of memory systems. The knowledge of the expert is represented in the form of metamorphic relations, which are properties of the analysed system involving multiple inputs and their outputs. Thus, the main contribution of this work is two-fold: a method to automatise the testing process of memory systems, and a novel expert system design focusing on increasing the overall performance of the testing process.

To show the applicability of our system, we have performed a thorough evaluation using 500 memory configurations and 4 different memory management algorithms, which entailed the execution of more than one million of simulations. The evaluation used mutation testing, injecting faults in the memory management algorithms. The developed expert system was able to detect over 99% of the critical injected faults, hence obtaining very promising results, and outperforming other standard techniques like random testing.

1. Introduction

Currently, CPU manufacturers are proposing multi-core CPUs as the answer to scaling up system performance. This trend has led to the emergence of increasingly powerful systems, provided with several CPUs consisting of multiple processing cores. As an example, the Multi-Purpose Processing Array processor integrates 256 cores in a single 28nm CMOS chip (de Dinechin, van Amstel, Poulhies, & Lager, 2014).

Systems based on multi-core architectures achieve a fair improvement level in terms of performance (Gepner & Kowalik, 2006). Generally, in multi-core platforms, the main memory is shared to enable communication between different processes (Mahapatra & Venkatrao, 1999). Hence, when the number of CPU cores increases, this memory becomes a system bottleneck and, consequently, emphasises the significance of *memory wall* (Wulf & McKee, 1995) and *bandwidth wall* (Rogers et al., 2009) problems. The current trend focuses on designing hierarchical multi-channel architectures aimed at exploiting the parallelism in multi-core systems. These architectures provide sophisticated and complex memory systems that alleviate this issue.

However, designing an efficient memory hierarchy is a difficult task faced by system designers. From a design perspective, there is a wide spectrum of possible configurations and parameters along multiple dimensions that must be carefully analysed before providing a valid design. For instance, there are different important key factors that have a direct impact on the overall memory performance. These include the number of memory controllers, their placement and the number of channels supported by each

* Corresponding author.
E-mail addresses: pablocc@ucm.es (P.C. Cañizares), alberto.nunez@pdi.ucm.es (A. Núñez), juan.delara@uam.es (J. de Lara).

https://doi.org/10.1016/j.eswa.2019.04.070
0957-4174/

controller, just to name a few (Abts, Jerger, Kim, Gibson, & Lipasti, 2009; Awasthi, Nellans, Sudan, Balasubramonian, & Davis, 2010; Kim, Han, Mutlu, & Harchol-Balter, 2010). Moreover, there is a vast number of choices related to the organisation of the memory device, like the hierarchical organisation of channels, banks, rows and columns (Jacob, Ng, & Wang, 2007).

Thus, checking the correctness of memory systems is crucial to ensure system scalability. Beyond the architectural design, testing scheduling policies orchestrated by the memory controller is a challenging problem. First, the controller needs to obey all DRAM timing constraints to provide correct functionality. Second, the controller must intelligently prioritise DRAM commands from different memory requests to optimise system performance (Ipek, Mutlu, Martínez, & Caruana, 2008). In order to completely check a memory system, in a systematic and exhaustive way, a broad range of hardware configurations and controller architectures must be analysed, which becomes a time-expensive and costly task. Moreover, analysing the power consumption of different memory chips may require an additional hardware, which significantly increases the monetary cost of the testing process.

Unfortunately, applying conventional testing techniques for checking memory architectures entails two main difficulties. First, test suites consisting of a large number of test cases are needed to accurately check the system under study, which requires a high computational cost. Additionally, since each test case must be executed in the platform under study, this process requires access to specific hardware. Second, an oracle that indicates if a given system is correct or not is, in most situations, unavailable or computationally too expensive (Weyuker, 1982). Moreover, the PASS/FAIL output provided by the major part of the testing techniques is not enough to locate where the fault has been produced. It is therefore desirable that the user obtains information that helps identifying the part of the system under test that is not working as expected.

In order to alleviate these issues, we propose an expert system that combines two orthogonal techniques to check the correctness of memory systems: simulation and metamorphic testing (in short, MT). On the one hand, simulation techniques are especially useful when the expected architectures are not available or expensive. Hence, these techniques provide a cost-effective method to simulate the behaviour of the target architecture. On the other hand, MT (Chen, Cheung, & Yiu, 1998) is a technique developed for testing systems where there is no oracle, or it is too expensive to compute (Weyuker, 1982). MT is based on metamorphic relations (in short, MRs), which describe properties of the system under study. The essential idea is that instead of checking the output o_1 produced when testing with one input x_1, we test with a second (follow-up) input x_2, observing the obtained output o_2, and then check that o_1 and o_2 are related as specified in the MR. Thus, in MT there are two relations: the relation between the original test input x_1 and the follow-up input x_2, and the expected relation between the two outputs.

Our proposed expert system consists of an inference engine, a knowledge base, a simulation platform, a graphical user interface and a factual database. The knowledge is introduced in the system by an expert in the form of MRs, which model the behaviour of the different parts of the memory system. The proposed system is scalable in the sense that the knowledge base can be updated by the expert, that is, the number of MRs can be increased, which improves the completeness of the system to check new memory models. Since each MR focuses on a specific part of the memory, the expert system is able not only to detect if a memory system is correct or not, but to provide precise information about the cause of the error and the part of the system where it is located. Additionally, the factual database stores the results of previous simulations, which increases the overall system performance by accessing those results that have been previously calculated.

In order to demonstrate both the applicability and suitability of the expert system, we have performed an experimental study using 500 different memory configurations and 4 memory management algorithms. In this study, we used 50 different workloads inspired by the PARSEC benchmarks (Bienia, Kumar, Singh, & Li, 2008). We designed 10 different MRs to analyse the correctness of each memory system configuration. In addition, 5 different mutants were generated for each scheduling algorithm to evaluate the effectiveness of our expert system. In the testing process we execute 25,000 different test cases over the original system and the generated mutants. Overall, in this process, we executed more than a million of simulations. The results obtained are promising, since the expert system was able to detect the vast majority of the injected faults. In general, 90% of the injected faults representing critical errors in the system have been detected, while the system provides acceptable results to detect those faults focusing on general aspects of the system, obtaining a 64% of average effectiveness. However, it is important to remark that when the expert combines several MRs in the knowledge base, the system is able to detect over 99% of the injected faults. Finally, a second set of experiments shows the benefits of our method with respect to standard techniques like random testing. The benefits are both in terms of effectiveness of the testing process (our method discovers more faults, more efficiently), and effort (the tester needs to provide a complete oracle, which is not required in our case).

The rest of the paper is organised as follows. Section 2 presents a literature review. Section 3 shows a brief overview of memory systems and introduces the main concepts of MT. In Section 4 we describe our model to represent different memory system configurations. Section 5 presents our catalogue of MRs. In Section 6, our proposed expert system is described in detail. Next, in Section 7, we present a thorough experimental evaluation. Finally, Section 8 finishes with the conclusions and future work.

2. Literature review

During the last years, several approaches targeted to check the correctness of memory systems have been proposed (Dreibelbis, Barth, Kalter, & Kho, 1998; Huang, Huang, Wu, Wu, & Chang, 1999; Karpovsky, van de Goor, & Yarmolik, 1995; Miyano, Sato, & Numata, 1999; Pundir & Sharma, 2017; Yang et al., 2015). Those proposals focus on creating fault models and testing algorithms to analyse different types of memory systems. Among them, we can highlight several significant techniques, such as design-for-testability (Dreibelbis et al., 1998), built-in self-test (Huang et al., 1999; Yang et al., 2015), simulation-based algorithms (Wu, Huang, Cheng, & Wu, 2000) and mathematical approaches using finite-state machines (de Goor & Smit, 1994a; 1994b). Although they are considered cost-effective to identify some of the most common errors in memory systems, none of them focus on memory management policies.

In the field of memory controllers, some novel techniques aimed to design and check new algorithms to enhance the scheduling process have been proposed. In most cases, these proposals are validated using manual and random testing techniques (Hassan, Patel, & Pellizzoni, 2015; Rixner, Dally, Kapasi, Mattson, & Owens, 2000), as well as with benchmarks (Ghasempour, Jaleel, Garside, & Luján, 2016; Lin, Reinhardt, & Burger, 2001). Random testing is a widely used technique for checking the correctness of computer systems. The main advantage of random testing is its simplicity, which allows to automatically generate test cases and execute them with a reasonable effort. However, this technique has some weaknesses. First, large test suites are required to reach a good level of coverage. Second, high computational resources are required to execute large test suites. Third, due to the stochastic nature of the generated test cases, there is no guarantee that these

are appropriate for accurately testing the system. Consequently, it is possible that critical parts of the system remain untested. On the contrary, manual testing is an arduous and error prone task that requires a considerable effort from the tester. The main advantage of this technique is the certainty that specific and critical parts of the system are tested. A common weakness of both techniques is the need for an oracle that checks if the outputs generated by the test cases are correct. In many cases, the (human) tester acts as the oracle.

In order to alleviate these issues, several proposals based on model checking (Clarke, Grumberg, & Peled, 2001) have been proposed. Sahoo and Satpathy proposed MSimDRAM (Sahoo & Satpathy, 2016), a formal model-driven framework to model and check DRAM controllers. The authors modelled the DRAM memory controller using the SAL language and finite state machines. In addition, in order to check the requirements, the authors encoded the correct behaviour using linear temporal logic (LTL). Then, they used bounded model checking, a technique for checking the satisfiability of a property. Khalifa and Salah presented a generic universal memory controller (Khalifa & Salah, 2015). This approach is based on a system-level architecture that has been verified using the universal verification methodology. The verification environment consists of two monitors, a reference transaction level model (TLM), and a driver that generates test cases, which are applied to the reference and the design under test to find possible errors. Kayed et al. presented a novel approach based on the JEDEC standards (Standard, 2012). This way, to verify the validity of the generated commands, the timing constraints are defined using a Timing Diagram Mark up Language and transformed into system Verilog assertions (Kayed, Abdelsalam, & Guindi, 2014). Li et al. proposed the modelling and verification of dynamic command scheduling for real-time memory controllers (Li, Akesson, Lampka, & Goossens, 2016). In this work, the memory controller is modelled using timed automata and is analysed using model checking through the UPAAL tool. Hassan and Patel proposed an approach to automate the validation process of DRAM memory controller designs, known as MCXplore (Hassan & Patel, 2017). This proposal provides a methodology oriented to generate test cases, using the properties defined in each policy, to maximise the coverage. Moreover, this methodology can be used to seamlessly validate the policies of memory controllers.

Although there are numerous advantages related to the use of model checking for analysing the correctness of systems, like the high coverage it achieves, the previously described approaches entail different issues, which are alleviated by our proposed system. First, these solutions are focused on verifying memory controllers with an ad-hoc model design and, therefore, these require specific requirements for each design, like a reference TLM model (Khalifa & Salah, 2015) and a specific register-transfer level implementation (Kayed et al., 2014). Hence, in order to successfully achieve a new version of the scheduler, several temporal logic constraints must be re-adapted and re-written for each memory controller. Second, in general, the translation of the system under test to a checkable-model using formal languages is a complex and difficult task. In some cases, the real system is too complex to be represented with enough fidelity using a given formalism. Third, these approaches are not scalable and require powerful resources and a high execution time. Finally, in some of these systems the user obtains a YES/NO answer indicating whether a system is correct or not and, therefore, lacking information to locate the anomaly. In summary, these facts hamper the generalisation of the model checking proposals to analyse memory management policies in real systems.

In the last 5 years, MT has been applied in different application domains, including the validation of complex systems (Segura, Fraser, Sanchez, & Ruiz-Cortés, 2016). For example, Jiang et al. proposed several MRs to ensure the correctness of CPU schedulers (Jiang, Chen, Kuo, & Ding, 2013). As a result, two faults were detected in one of the simulators under study. Ding et al. presented an iterative approach focused on the development and refinement of MRs for testing scientific applications (Ding, Zhang, & Hu, 2016). Núñez and Hierons proposed a methodology based on MT for fault detection in cloud systems (Núñez & Hierons, 2015). Cañizares et al. proposed preliminary ideas for designing energy aware systems (Cañizares, Núñez, Núñez, & Pardo, 2015). These works show that MT can be applied in complex systems where an oracle is not available.

During the last decade, simulation tools have gained popularity to model and analyse memory systems. Rosenfeld et al. presented DRAMSim2, whose main strength is its accuracy for simulating DDR2/DDR3 models (Rosenfeld, Cooper-Balis, & Jacob, 2011). Also, DRAMSim2 provides different models to represent the energy consumption of the simulated memories. The main weakness of DRAMSim2 is the lack of mechanisms to deploy different memory management policies. The Utah SImulated Memory Module (in short, USIMM (Chatterjee et al., 2012)) is a trace-based memory system simulator focused on DDRx memories. USIMM provides mechanisms for modelling the different components of the memory system, such as the system architecture, DRAM timing and latency parameters, scheduling policies and power consumption. Moreover, USIMM includes some of the most used PARSEC benchmarks (Bienia et al., 2008). Jeong et al. proposed DrSim, a simulation platform for modelling DRAM systems, which provides a widespread spectrum of memory architectures and topologies (Jeong, Yoon, & Erez, 2012). Similarly, Kim et al. (Kim, Yang, & Mutlu, 2016) presented Ramulator, a fast and cycle-accurate DRAM simulator that supports an extensive spectrum of DRAM standards, such as DDR3/4, LPDDR3/4, GDDR5, WIO1/2 and HBM. The main advantage of this simulator is its performance, which appoints Ramulator as the fastest memory simulator. However, several weaknesses like the high abstraction level of the memory components and the lack of both power consumption models and memory management policies, make Ramulator not appropriate for our proposed system. Although these simulators support modelling and simulation of memory systems, the testing process must be manually performed.

To the best of our knowledge, expert systems have not been applied to check the correctness of memory systems. However, they have been successfully applied – as an effective approach for analysing complex systems – to a wide variety of domains including, among others, acoustic diagnosis (Hussain, S.J. Lee, M.S. Choi, & Brikci, 2015), power systems (Liberado, ao, oes, W.A. de Souza, & Pomilio, 2015), geographic information systems (C.M. Herrero-Jiménez, 2012), fault diagnosis of computer systems (Bennett & Hollander, 1981) and productivity of industrial environments (J. Bautista-Valhondo & R. Alfaro-Pozo, 2018). Hence, we think that expert systems are perfectly suitable to achieve our goals.

3. Background

In this section we provide introductory concepts about the memory system and MT.

3.1. The memory system

Since the last decades, memory systems are based on Dynamic Random Access Memory technologies (in short, DRAM) (Hardee, Chapman, & Pineda, 1991). The continuous evolution of the computational systems has encouraged the sophistication of this technology by increasing its throughput and capacity. Currently, DDR4 is the prevailing off-chip memory technology. The initial JEDEC DDR4 DRAM specification was released in September

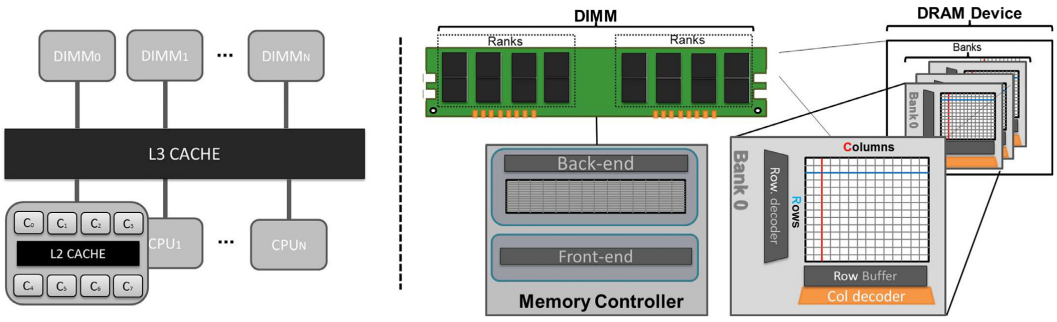


Fig. 1. Architecture of the memory system.

2012 (Council, 2017), where memory speeds described in this standard are expected to reach 3,200 Mbps, while its predecessor DDR3 reached 1,600 Mbps (Mukundan, Hunter, hyoun Kim, Stuecheli, & Martínez, 2013).

The system organisation and main components of DRAM are illustrated in Fig. 1. The left of the figure depicts a typical high-performance processing architecture. In this environment, several processors are connected to the memory system through a memory controller, which allows parallel access using multiple channels.

Current DDRx memories are structured as Dual In-line Memory Modules (in short, DIMM), which consist of several devices. The DIMMs are interconnected through a data bus to the memory controller. The right part of Fig. 1 shows the DRAM channel organisation, which consists of a set of channels connected to a collection of DIMMs. Each DIMM consists of a small number of *ranks*, which contain several DRAM devices (also called chips).

A rank is a collection of DRAM devices that operate in parallel. Thus, each *rank* is itself partitioned into a set of *banks* that are independently controlled and have their own row buffers (also called pages) of data (Sudan et al., 2010). The DRAM controller manages the memory requests using several scheduling policies, while obeying timing and hardware constraints of the DRAM chip to improve performance.

Requests from the CPU arrive to the memory controller, which translates them into a collection of orchestrated commands for DRAM access. Once the data request arrives to the controller, the memory address is extracted and then the channel is selected. At this point, it is important to avoid information leakage in the memory controller (Gundu et al., 2014; Shafiee, Gundu, Shevgoor, Balasubramonian, & Tiwari, 2015). Next, the DIMM and a rank inside it are calculated. Thus, DRAM devices within a rank synchronously work together to return as many bits of data as the width of the channel. In general, accesses to a DRAM device require first selecting a bank and then a row. For any read request, a row of data is read into the row-buffer associated with the bank.

3.2. Metamorphic testing

Traditional testing techniques require checking the conformance between the input(s) and the output(s) of the system under study. Schematically, let $\mathcal{S}$ be a system, I the input domain and TS a test selection strategy. Let $\mathcal{T} = \{t_1, t_2, \ldots, t_n\} \subseteq I$ be the set of tests generated by using TS. When these tests are sequentially applied to the system $\mathcal{S}$ we obtain a sequence of outputs $\mathcal{S}(t_1), \mathcal{S}(t_2), ..., \mathcal{S}(t_n)$. Given an oracle f, an error is found in $\mathcal{S}$ if there exists $t_i \in \mathcal{T}$ such that $\mathcal{S}(t_i) \neq f(t_i)$.

However, a complete oracle f, able to exactly characterise the expected output of a test, is challenging in many domains, including ours. In order to alleviate this problem, we propose using MT techniques (Chen et al., 1998; Ding et al., 2016; Liu, Kuo, Towey, & Chen, 2014). The main difference between traditional testing techniques and MT lies in the comparison of the obtained outputs. This way, while traditional techniques compare the output of each individual test case with the one obtained from the oracle, MT checks the relation between multiple test inputs and their outputs.

MT uses expected properties of the target system relating multiple test inputs with the corresponding outputs obtained from the system under test. These properties are formulated as metamorphic relations. A metamorphic relation (in short, MR) is a property of the analysed system that involves multiple inputs and their outputs. We represent a MR as a tuple (MR_i, MR_o), where MR_i refers to the relation between the source test case and the follow-up test case, and MR_o refers to the relation that must be fulfilled by the outputs obtained from the source test case and the follow-up test case.

Fig. 2 illustrates the MT process. Initially (activity with label ①), the tester must build a repository of MRs, which act as an oracle to check whether the outputs returned by the system under test are the expected ones. These MRs must be designed according to the specification of the system under test. In this work, the specification of memory systems is used to create MRs. Please note that the tester must be able to interpret the specification of the system under test, if it is available, to properly build suitable MRs. This task is specially challenging when the system under study is complex.

Next, for each *MR* in the repository of MRs, the tester must build a test suite consisting of a collection of source test cases. These test cases must be generated considering the specification of the system under test (label ② in the Figure). Any traditional testing technique, like random testing (Ciupa, Pretschner, Oriol, Leitner, & Meyer, 2011), can be used to create each test suite. Similarly, in the next step (label ③), for each previously generated source test suite in ②, a new follow-up test suite, containing the same number of test cases, is built. Thus, follow-up test cases are generated by using both the source test cases and the input relation of *MR*, that is, MR_i.

Next, each test case generated in the previous steps is executed against the system under study (label ④). When the execution of all the test cases finishes, the MRs are used to check the obtained outputs (label ⑤). In order to accomplish this task, these MRs are chosen one by one from the repository of MRs created in ①. Hence, for each *MR*, the source and follow-up test cases are used to check whether their outputs satisfy the relation given by MR_o. If the relation is not satisfied, an error has been found, and the corresponding fault and the violated MR are stored to analyse

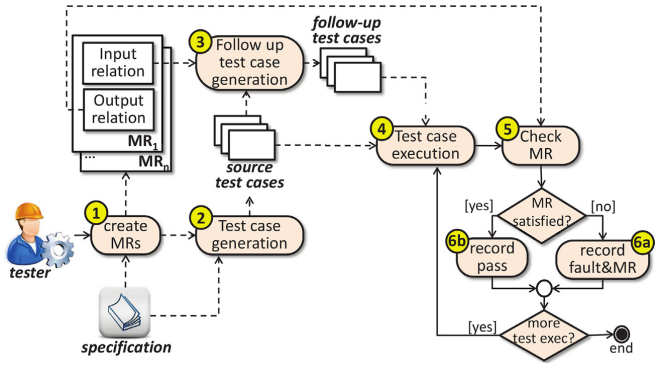


Fig. 2. Scheme of the MT process.

the issue (see 6a). On the contrary, if the *MR* is fulfilled, the statistics are updated increasing the number of test cases that satisfy this relation (see 6b). Once all the test cases are checked against the *MR*, the next MR is chosen. This process is repeated until all the MRs are processed.

4. Modelling the memory system and the workloads

Our approach needs a model of the memory system, and for this purpose we use the structure given by Definition 1. This structure is inspired by the architecture of DDRx systems, where off-chip memories associated with each memory channel are hierarchically organised in terms of ranks, banks, rows and columns (Jacob et al., 2007). Also, the memory page can be modelled by configuring the row buffer size parameter (in short, *rbs*). In order to provide a high level of flexibility, this model supports the configuration of a variable number of latencies. Thus, a wide spectrum of memory systems, using different levels of complexity, can be modelled. It is important to remark that our proposed expert system is not focused on a specific simulation platform. On the contrary, the user must select the most appropriate parameters used in the target simulator to configure the required memory system from the model.

Definition 1. A memory system m is a tuple (*dimm, chan, rank, chip, bank, row, col, freq, rbs, lat*), where:

- $dimm \in \mathbb{N}$ is the number of memory modules,
- $chan \in \mathbb{N}$ is the number of channels,
- $rank \in \mathbb{N}$ denotes the set of DRAM chips that can be simultaneously accessed,
- $chip \in \mathbb{N}$ is the number of DRAM chips of each *rank*,
- $bank \in \mathbb{N}$ refers to the total number of banks in each *chip*,
- $row \in \mathbb{N}$ represents the total number of rows per *bank*,
- $col \in \mathbb{N}$ represents the total number of columns per *bank*,
- $freq \in \mathbb{R}$ denotes the frequency of m measured in MHz,
- $rbs \in \mathbb{N}$ denotes the row buffer size measured in bytes, and
- *lat* is a set $\{l_i | i \geq 0\}$ that refers to the different latencies used in m.

We denote by m_{size} the total size of the memory system m, measured in bytes, which can be calculated using the following formula:

$$m_{size} = m_{dimm} * m_{rank} * m_{chip} * m_{bank} * m_{row} * m_{col} \quad (1)$$

The input tests are described by workloads, which are sequences of operations that are executed over a memory system. Their structure is given by Definition 2.

Definition 2. A workload ω is a sequence $\{r, w\}^*$, where:

- r represents a read operation, denoted as a tuple (*nops, addr*1, *addr*2) where *nops* is the number of non-memory instructions carried out before the operation, *addr*1 and *addr*2 are the addresses related with the read operation.
- w is a write operation, denoted as a tuple (*nops, addr*) where *nops* is the number of non-memory instructions before the operation and *addr* is the address where the write is performed.

In the following, we denote the empty sequence by ϵ, and use $numR(\omega)$ and $numW(\omega)$ to refer to the number of read and write operations in the workload ω. These operations can be calculated using the following functions:

$$numR(\omega) = \begin{cases} 0 & \text{if } s = \epsilon \\ 1 + numR(\omega') & \text{if } s = r \cdot s' \end{cases}$$

$$numW(\omega) = \begin{cases} 0 & \text{if } s = \epsilon \\ 1 + numW(\omega') & \text{if } s = w \cdot s' \end{cases}$$

We say that a workload ω is included in ω', written $\omega \leq \omega'$, if $\exists x, y \in \{r, w\}^*$ s.t. $\omega' = x \cdot \omega \cdot y$. We write $\omega = \omega'$ when both workloads are equal, that is, ω and ω' have the exact same elements in the same order.

In order to test a memory system, a suitable collection of test cases needs to be generated. A test case is a pair (m, ω^n), where m is a memory system, ω is a workload and n is the number of workload instances to be executed. These instances are equal, that is, each one has the exact same elements. For the sake of clarity, we use the notation ω (omitting the super-index) to represent the execution of 1 workload instance. We assume a suitable simulator able to run the workloads, and produce information about time, power, and number of performed read and write operations.

Definition 3. Let m be a memory system and ω be a workload. The result of simulating the execution of n workload instances of ω over the memory system m is denoted by $S(m, \omega^n)$. In those cases where $n > 1$, we assume that a dedicated CPU is used to execute each workload instance.

The output obtained from simulating the workload ω over the memory system m is represented with the following notation:

- $S_T(m, \omega^n) \in \mathbb{R}_+$ denotes the time required to execute n instances of ω over m.
- $S_P(m, \omega^n) \in \mathbb{R}_+$ denotes the power required to execute n instances of ω over m.

- $S_W(m, \omega^n) \in \mathbb{N}_0$ denotes the number of performed write operations to execute n instances of ω over m.
- $S_R(m, \omega^n) \in \mathbb{N}_0$ denotes the number of performed read operations to execute n instances of ω over m.

5. A catalogue of MRs for testing memory systems

In this work we have designed a suitable collection of MRs[1]. These relations represent properties of the system under test – inferred by experts – that are stored in the knowledge base and used by the inference engine to check the correctness of memory systems. Next, we formally define the pattern of our relations.

Definition 4. A *metamorphic* relation *MR* for a memory system m and a workload ω is the set of 4-tuples

$$MR(m,\omega) = \left\{ \begin{pmatrix} (m,\omega), \\ (m',\omega') \\ S(m,\omega), \\ S(m',\omega') \end{pmatrix} \middle| \begin{array}{c} MR_i((m,\omega),(m',\omega')) \\ \Downarrow \\ MR_o(S(m,\omega),S(m',\omega')) \end{array} \right\}$$

where MR_i is a relation over the source test case and a follow-up test case, and MR_o is a relation over the results obtained from the execution of these test cases.

In this work we propose a catalogue of MRs focusing on different aspects of memory systems, such as performance, energy consumption and functionality. The design of these MRs is inspired by common errors found in memory management algorithms. In order to identify these errors, different sources have been carefully investigated. We started our efforts by analysing bug reports and "*whats new*" logs from different repositories containing memory management algorithms. In particular, we analysed repositories of well-known simulators, including Ramulator (Kim, 2017), DRAMSim (Rosenfeld, 2017), NVMain (Poremba, 2017) and USIMM (Kumar, 2017). From this analysis we gathered several errors committed by real programmers, like wrong managing of ranks, wrong use of channel indexes, wrong timing in write operations, deadlock scenarios with large memory sizes and several issues with the memory controller state, among others.

Next, we studied different papers found in the current literature. Since memory scheduling is the most important function of the memory controller, the research community has invested a significant effort to analyse and test a wide-spectrum of techniques to optimise the overall performance of memory systems (Modgil, Nitin, & KumarSehgal, 2015; Natarajan, Christenson, & Briggs, 2004). We focused our efforts in investigating those aspects of the analysed techniques that may produce a bug in the system like, just to name a few, delayed write scheduling, request re-ordering features and in-order request processing.

Finally, we studied a specialised site on micro-controller architectures. Specifically, we investigated the topic focusing on software-based memory testing (Barr, 2017), where the author remarks the importance of detecting issues in the scheduling algorithms to avoid catastrophic failures, like bypassing a memory channel or bypassing a rank.

In the following, we describe the catalogue of the 10 proposed MRs. These focus on checking for errors gathered from the previous study.

$$MR_1 = \left\{ \begin{pmatrix} (m,\omega), \\ (m',\omega') \\ S(m,\omega), \\ S(m',\omega') \end{pmatrix} \middle| \begin{array}{c} \omega = \omega' \wedge m_{chan} > m'_{chan} \\ \wedge \\ m_{lat} = m'_{lat} \\ \Downarrow \\ S_T(m,\omega) \leq S_T(m',\omega') \end{array} \right\}$$

[1] The collection of MRs is available at http://antares.sip.ucm.es/cana/MRlist.pdf

***MR*$_1$:** Given two memory models m and m' and two workloads ω and ω', if the workloads ω and ω' is equal, and the number of channels of m is greater than the number of channels of m', and the latencies of both memories are equal, then the time required to execute ω over m should be less or equal than the one required to execute ω' over m'.

$$MR_2 = \left\{ \begin{pmatrix} (m,\omega), \\ (m',\omega') \\ S(m,\omega), \\ S(m',\omega') \end{pmatrix} \middle| \begin{array}{c} m = m' \\ \wedge \\ numW(\omega) > numW(\omega') \\ \wedge \\ numR(\omega) > numW(\omega') \\ \Downarrow \\ S_T(m,\omega) > S_T(m',\omega') \end{array} \right\}$$

***MR*$_2$:** Given two memory models m and m' and two workloads ω and ω', if the memory systems m and m' are equal, and the number of write operations in ω is greater than the number of write operations in ω', and the number of read operations in ω is greater than the ones of ω', then the time required to execute ω over m should be greater than the time required to execute ω' over m'.

$$MR_3 = \left\{ \begin{pmatrix} (m,\omega), \\ (m',\omega') \\ S(m,\omega), \\ S(m',\omega') \end{pmatrix} \middle| \begin{array}{c} \omega = \omega' \wedge m_{size} < m'_{size} \\ \Downarrow \\ S_W(m,\omega) = S_W(m',\omega') \\ \wedge \\ S_R(m,\omega) = S_R(m',\omega') \end{array} \right\}$$

***MR*$_3$:** Given two memory models m and m' and two workloads ω and ω', if ω and ω' are equal and the size of memory m is smaller than the size of memory m', then the number of performed write and read operations during the execution of ω over m should be equal to the number of performed write and read operations during the execution of ω' over m', respectively.

$$MR_4 = \left\{ \begin{pmatrix} (m,\omega), \\ (m',\omega') \\ S(m,\omega), \\ S(m',\omega') \end{pmatrix} \middle| \begin{array}{c} \omega = \omega' \\ \wedge \\ m_{lat} \geq m'_{lat} \wedge m_{chan} = m'_{chan} \\ \Downarrow \\ S_T(m,\omega) \geq S_T(m',\omega') \end{array} \right\}$$

***MR*$_4$:** Given two memory models m and m' and two workloads ω and ω', if the workloads ω and ω' are equal and the latency of memory m is greater or equal than the latency of memory m', and the number of channels of both memories are equal, then the time required to execute ω over m should be greater than or equal to the time required to execute ω' over m'.

$$MR_5 = \left\{ \begin{pmatrix} (m,\omega), \\ (m',\omega') \\ S(m,\omega), \\ S(m',\omega') \end{pmatrix} \middle| \begin{array}{c} \omega = \omega' \\ \wedge \\ m_{lat} \geq m'_{lat} \wedge m_{chan} = m'_{chan} \\ \Downarrow \\ S_W(m,\omega) = S_W(m',\omega') \\ \wedge \\ S_R(m,\omega) = S_R(m',\omega') \end{array} \right\}$$

***MR*$_5$:** Given two memory models m and m' and two workloads ω and ω', if ω and ω' are equal and the latency of memory m is greater than or equal than the latency of memory m', and the number of channels of both memories are equal, then the number of performed write and read operations during the execution of ω over m should be equal to the number of performed write and read operations during the execution of ω' over m', respectively.

$$MR_6 = \left\{ \begin{pmatrix} (m,\omega), \\ (m',\omega') \\ S(m,\omega), \\ S(m',\omega') \end{pmatrix} \middle| \begin{array}{c} m = m' \wedge \omega' \leq \omega \\ \Downarrow \\ S_P(m,\omega) \geq S_P(m',\omega') \end{array} \right\}$$

***MR*$_6$:** Given two memory models m and m' and two workloads ω and ω', if the memory systems m and m' are equal and all the

elements of ω' are contained in ω, then the power required to execute ω over m should be greater than or equal to the power required to execute ω' over m'.

$$MR_7 = \left\{ \begin{pmatrix} (m,\omega), \\ (m',\omega') \\ S(m,\omega), \\ S(m',\omega') \end{pmatrix} \middle| \begin{array}{c} \omega = \omega' \wedge m_{size} < m'_{size} \\ \wedge \\ m_{lat} = m'_{lat} \wedge m_{chan} = m'_{chan} \\ \Downarrow \\ S_P(m,\omega) < S_P(m',\omega') \end{array} \right\}$$

***MR*$_7$:** Given two memory models m and m' and two workloads ω and ω', if the workloads ω and ω' are equal and the size of memory m is less than the size of memory m', and the latencies of both memories are equal, and both memories have the same number of channels, then the power required to execute ω over m should be less than the power required to execute ω' over m'.

$$MR_8 = \left\{ \begin{pmatrix} (m,\omega), \\ (m',\omega'^n) \\ S(m,\omega), \\ S(m',\omega'^n) \end{pmatrix} \middle| \begin{array}{c} m = m' \wedge \omega = \omega' \wedge n > 1 \\ \Downarrow \\ \lceil \frac{n}{m_{chan}} \rceil \cdot S_T(m,\omega) \\ > \\ S_T(m',\omega'^n) \end{array} \right\}$$

***MR*$_8$:** Given two memory models m and m' and two workloads ω and ω', if the memory systems m and m' are equal, and the workloads ω and ω' are equal, then the time required to execute n instances of ω' over m' should be less than $\lceil \frac{n}{m_{chan}} \rceil$ times the time required to execute ω over m.

$$MR_9 = \left\{ \begin{pmatrix} (m,\omega), \\ (m',\omega'^n) \\ S(m,\omega), \\ S(m',\omega'^n) \end{pmatrix} \middle| \begin{array}{c} m = m' \wedge \omega = \omega' \wedge n > 1 \\ \Downarrow \\ n \cdot S_W(m,\omega) = S_W(m',\omega'^n) \\ \wedge \\ n \cdot S_R(m,\omega) = S_R(m',\omega'^n) \end{array} \right\}$$

***MR*$_9$:** Given two memory models m and m' and two workloads ω and ω', if the memory systems m and m' are equal and the workloads ω and ω' are equal, then the number of performed write and read operations during the execution of n instances of ω' over m' should be equal to n times the number of performed write and read operations during the execution of ω over m, respectively.

$$MR_{10} = \left\{ \begin{pmatrix} (m,\omega), \\ (m',\omega'^n) \\ S(m,\omega), \\ S(m',\omega'^n) \end{pmatrix} \middle| \begin{array}{c} m = m' \wedge \omega = \omega' \wedge n > 1 \\ \Downarrow \\ S_P(m,\omega) < S_P(m',\omega'^n) \end{array} \right\}$$

***MR*$_{10}$:** Given two memory models m and m' and two workloads ω and ω', if the memory systems m and m' are equal and the workloads ω and ω' are equal, then the power consumption associated to execute ω over m should be less than the power required to execute n instances of ω' over m', it being $n > 1$.

6. Proposed expert system

An expert system is a computational system that emulates the decision-making of human expertise using domain specific knowledge. The main difference between an expert system and a conventional one lies in the method used to solve complex problems. That is, while expert systems apply reasoning based on rules, conventional systems are based on procedural code.

In this section we present our proposed expert system. Its basic architecture is shown in Section 6.1 and the testing procedure is described in Section 6.2.

6.1. Architecture of the proposed expert system

In this work we propose an expert system for checking the correctness of memory systems using MT and simulation techniques. In contrast with the conventional architecture of expert systems, we have included two additional modules: a factual database and a simulation platform. Thus, our proposed system consists of 5 main modules (see Fig. 3).

The *knowledge base* (in short, KB) is a module that is built using the knowledge of the expert. In essence, KB consists of rules and facts. In this case, the rules are introduced into the KB by the expert, in the form of MRs.

In some situations, different MRs may have the same MR_i (the relation with the source test case and the follow-up test case), while the MR_o (the relation that must be fulfilled by the obtained outputs) differs. In particular, this is the case of MR_4 and MR_5, where the former relation focuses on performance by comparing the time required to execute a given workload, while the latter focuses on functionality by comparing the number of read and writes. In these cases, there are two possible solutions. First, using two different MRs, like MR_4 and MR_5. This solution is used when different outputs, obtained from the same input, are required to be separately analysed. Second, to construct a new MR by combining several existing relations (Liu, Liu, & Chen, 2012). For instance, let us consider a memory scheduler that must process a given workload. The algorithm of this scheduler may, or may not, fulfil the corresponding specification. Also, this algorithm may, or may not, be efficient enough to be considered acceptable. If we use our proposed expert system to check this algorithm, we will be able to decide if there is an unexpected behaviour of the algorithm under test by analysing those tests that do not fulfil MR_4 and MR_5. Hence, if there are tests that do not fulfil MR_4, there is an error in the implementation of the algorithm. On the contrary, if there are tests that do not fulfil MR_5, we can assume that the obtained performance of the scheduling algorithm is not acceptable. However, if we use a combined MR, it is more difficult to locate the unexpected behaviour of the system under test.

In some cases using all the available MRs may be too expensive in terms of computational and time costs, and a subset or combination of them must be selected. This is so because the follow-up test cases must be created ad-hoc for each MR. Therefore, the higher the number of MRs, the higher the number of follow-up test cases that need to be generated. Consequently, the expert has to take the decision of introducing into the KB separate MRs or to combine them.

The *user interface* module is a friendly and easy-to-use application – written in Java – that provides a graphical user interface (in short, GUI). Using this GUI, non-expert users can perform different tasks like modelling new memory systems, editing the configuration of a current memory model and testing memory models. Fig. 4 shows the editor for modelling memory systems. Basically, this editor contains each parameter of the memory model and its corresponding value. The left part of the panel shows a repository of memory models, where users can easily save, edit and remove memory models in the application.

Fig. 5 shows a panel that allows users to select the MRs that will be used in the testing process. These rules are obtained from the KB and displayed in the GUI. Thus, if the expert updates the rules in the KB, these are also updated in the GUI.

The *simulation platform* (in short, SP) is in charge of two main tasks. First, once the non-expert user has defined a memory model and selected the required MRs, the SP uses this information to automatically generate a set of follow-up test cases. Second, for each generated follow-up test case, the *factual database* (in short, FDB) is accessed to request information of the test. If the test is stored in the FDB, then the required information is obtained from the SP. In other case, the SP executes the simulation of the test case to produce the results and to extract the required information to be stored in the FDB.

The facts and rules – also called MRs – are analysed by the *inference engine* (in short, IE) and, for each test case, the IE checks if

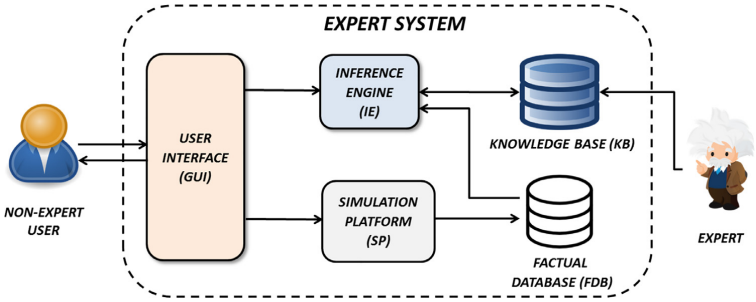


Fig. 3. Architecture of the proposed expert system.

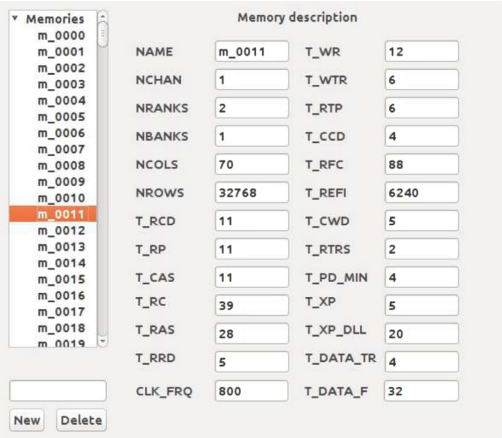


Fig. 4. GUI: Editor for modelling memory configurations.

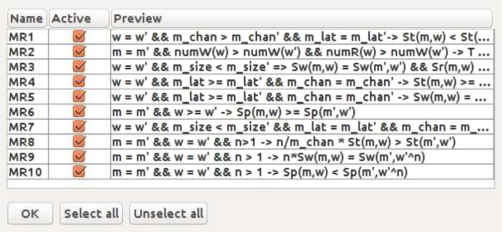


Fig. 5. GUI: Panel to select the MRs used in the testing process.

the involved MRs in the testing process are fulfilled by matching the obtained outputs.

6.2. Testing procedure

This section presents a detailed description of the required steps to test a memory system using our approach (see Fig. 6). For the sake of clarity and completeness, we also describe some relevant internal steps performed by the main modules of the expert system.

In the first place, a memory model needs to be defined (label 1). Next, the non-expert user must select a memory scheduling policy (label 2), and as in any testing process, a set of input test cases need to be provided (label 3). These test cases are inspired by the PARSEC suite (Bienia et al., 2008). In the following step (label 4) the user selects one, or several, MRs, and follow-up test cases are automatically generated by the SP (label 5).

$$
\begin{aligned}
l &= (9,9,9,39,28,5,32,12,6,6,\\
&\quad 4,128,6240,5,2,4,5,20,4)\\
m &= (1,\mathbf{2},7,32768,128.0,1)\\
\omega &= \{r,r,r,w,r,w,r\}\\
T_{source} &= (m,\omega)
\end{aligned}
$$

Listing 1. Source test case.

$$
\begin{aligned}
l' &= (9,9,9,39,28,5,32,12,6,6,\\
&\quad 4,128,6240,5,2,4,5,20,4)\\
m' &= (1,\mathbf{1},7,32768,128.0,l') \blacktriangleleft\\
\omega' &= \{r,r,r,w,r,w,r\}\\
T_{follow\text{-}up} &= (m',\omega')
\end{aligned}
$$

Listing 2. Follow-up test case matching MR1.

In order to illustrate the concepts described in this section, we present an example that shows the generation of a follow-up test case using MR_1. Listing 1 represents a source test case, where m is a memory system, l is a tuple representing the latencies of m and ω is a (simplified) workload. Listing 2 shows the generated follow-up test case, which satisfies the relation of MR_{1_i}. This test case has been generated by using a memory containing less channels than the memory used in the source test case. The workload and latencies are the same for both test cases.

In step 6, the SP checks if the required information for the test exists in the FDB. If this is the case, then the simulation of the test case is not executed, and this information is obtained from the FDB. On the contrary, the memory management policy is simulated on the memory model using both the input test cases and the follow-up test cases. The simulation provides outputs, typically informing about the consumed power, time, and number of read/write operations.

Next, the IE uses both the facts and rules from the KB and FDB to check those MRs that are fulfilled by the test cases. If they do not match, it means an error has been found. If they match, the confidence on the correctness of the memory system increases.

Finally, in the last step (label 8), the IE generates a report that is displayed in the GUI. Fig. 7 shows the results of testing a memory system containing faults. In this case, the expert system detects faults in the read queue, write queue and delays in the operations,

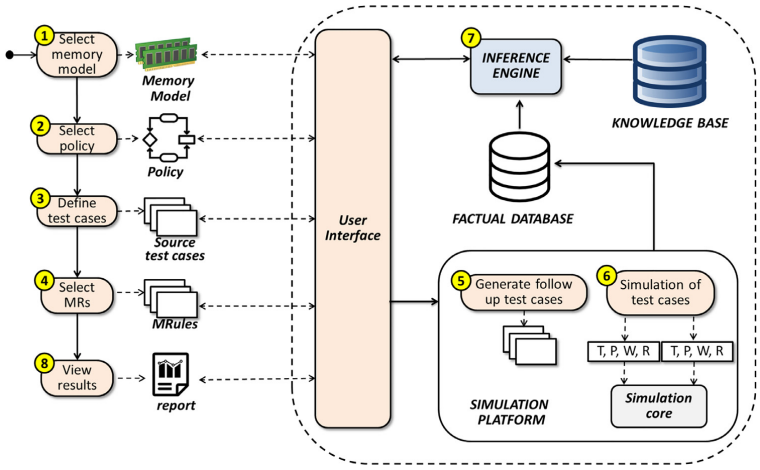


Fig. 6. Scheme of the testing procedure.

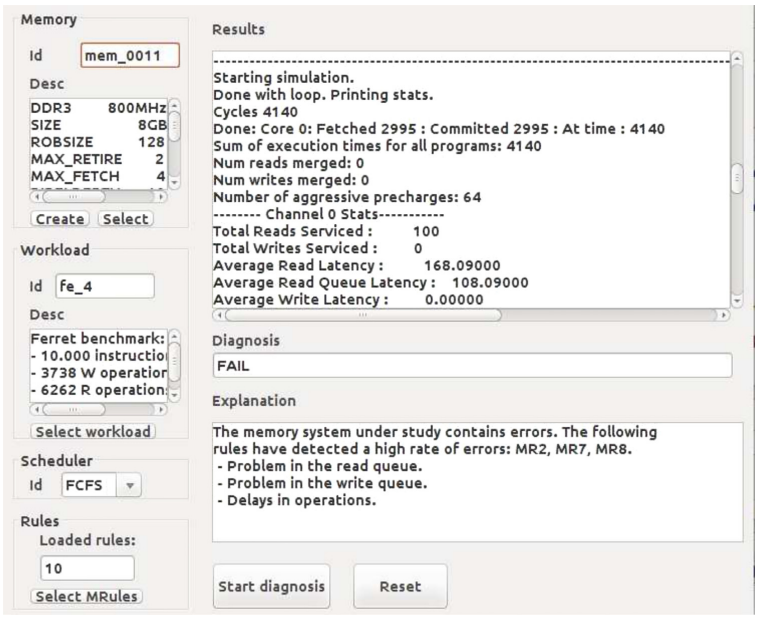


Fig. 7. GUI: Results of a faulty memory system.

which are detected by MR_2, MR_7 and MR_8 (see Explanation area in Fig. 7). Similarly, Fig. 8 shows the results of testing a correct memory system.

It is important to remark that our proposed expert system locates faults in *isolated* memory systems using simulation. Thus, a wide spectrum of memory configurations can be automatically and efficiently tested. If a fault is found during the testing process, the expert must fix the memory system module of the corresponding operating system. Unfortunately, not all the existing operating systems allow the modification of the memory system internals, like the memory allocating policy and, therefore, the applicability of our proposed system depends on the availability of the target operating system to apply changes in its memory system. However, open-source operating systems, like Linux, allow not only to modify the actual memory system configuration, but to include new and customised algorithms. In any case, using an appropriate and error-free memory system accurately exploits the overall system performance achieving, in the major part of the cases, a good trade-off between performance and hardware complexity (Modgil et al., 2015; Subramanian, Lee, Seshadri, Rastogi, & Mutlu, 2016).

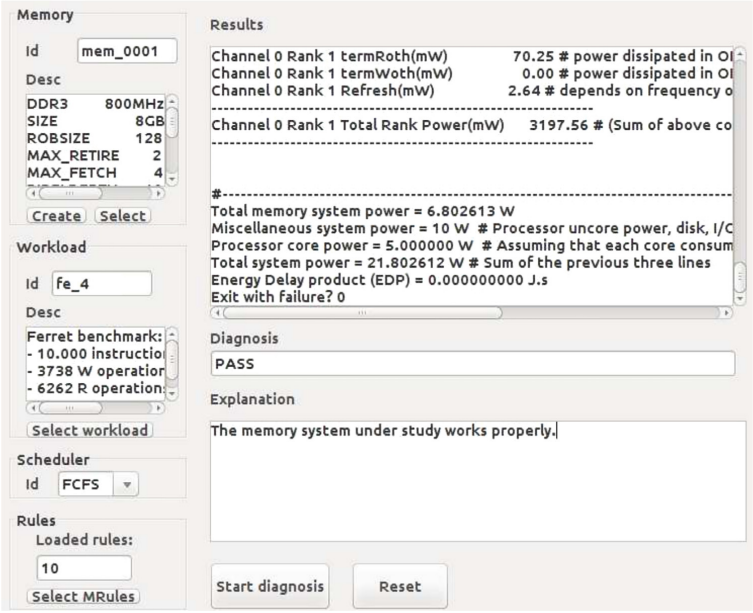


Fig. 8. GUI: Results of a correct memory system.

7. Empirical study

Once we have developed our expert system, there are three main research questions (RQs) to be answered, namely:

RQ1: *Are the designed MRs suitable to be used as rules in the knowledge base?*
We ask the first research question to check if the designed KB – using MRs – properly represents the underlying behaviour of the memory system. To answer **RQ1**, we have performed an experiment using the USIMM simulator (Chatterjee et al., 2012) (which we initially assumed correct), creating 500 memory models and 50 different workloads based on the PARSEC suite (Bienia et al., 2008), and using 4 memory management algorithms[2]. The results of these experiments are detailed in Section 7.2.

RQ2: *How effective is the proposed expert system to catch errors in faulty memory systems?*
Next, we ask a further question to know the effectiveness of the expert system for detecting errors in faulty memory systems. In this case, we answer **RQ2** by testing different faulty versions of the system under test. Hence, minor syntactical faults were artificially injected in the memory management system, which is considered the main core of the memory system. These faulty versions are known as *mutants* in the sense of *mutation testing* (Hierons, Merayo, & Núñez, 2010). The results of this experiment are described in Section 7.3.

RQ3: *How suitable is the proposed expert system to test memory systems compared with standard methods?*
Finally, we are interested in investigating the suitability of our expert system for testing memories, compared to other methods. In order to ask this question, we have analysed the current standard methods for testing memory systems. We focus the search on general methods that are suitable for analysing a wide spectrum of memory configuration. Hence, methods for testing a specific memory system are not considered in this study. The results of this comparison are shown in Section 7.4.

The rest of this section is organised as follows. The experimental setting is described in Section 7.1. Sections 7.2 and 7.3 detail the two experiments performed. A comparison between our approach and a standard method for testing memory systems is shown in Section 7.4. The results obtained in the experiments are discussed in Section 7.5, where we also answer the research questions. Finally, we analyse threats to validity in Section 7.6.

7.1. Experimental setting

The main goal of the proposed expert system is to check the correctness of memory systems using realistic memory models with a high level of detail. Hence, even though our system is general and independent of a specific simulator, we consider USIMM as the most suitable option for the purposes of our study. First, this simulator provides a good compromise between the high level of detail in the hardware models and the inherent flexibility to model a wide range of memory systems. Second, USIMM supports simulation using customised memory management policies. In fact, USIMM has been used in the Memory Scheduling Championship (Chatterjee et al., 2012). Third, this simulator provides an accurate power consumption model and generates a detailed collection of statistics as output.

For both experiments 500 different memory models have been generated. Also, 50 different workloads have been created, which are inspired by PARSEC benchmarks, such as *blackscholes, facesim, ferret, fluidanimate, freqmine, streamcluster and swaptions*. In order to generate traces that are representative of the selected

[2] The experiments results can be found at http://antares.sip.ucm.es/tools/expertSystems/experiments.7z

Table 1
Configuration parameters in USIMM (Chatterjee et al., 2012).

Parameter	Description
NUM_CHANNELS	Number of channels
NUM_RANKS	Number of ranks per channel
NUM_BANKS	Number of banks per chip
NUM_ROWS	Number of rows
NUM_COLUMNS	Number of columns
DRAM_CLK_FREQ	Frequency of the memory (MHz)
T_RCD	Row to Column interval delay
T_RP	Row pre-charge delay
T_CAS	Column access
T_RC	Row cycle interval delay
T_RAS	Row access strobe
T_RRD	Interval row activation delay
T_FAW	Four bank activation window
T_WR	Write recovery time
T_WTR	Write to read delay time
T_RTP	Read to pre-charge
T_CCD	Column to Column delay
T_RFC	Refresh cycle time
T_REFI	Refresh interval period
T_CWD	Column write delay
T_RTRS	Rank to Rank switching time
T_PD_MIN	Minimum power down duration
T_XP	Time to exit fast power down
T_XP_DLL	Time to exit slow power down
T_DATA_TR	CPU to memory transfer time

benchmarks, the SimPoint platform has been used (Sherwood, Perelman, Hamerly, Sair, & Calder, 2003). During the trace generation process, SimPoint uses basic block vectors to recognise execution intervals that can be used to reflect the behaviour of the benchmark.

Each memory model generated in this study has been tested with 4 different memory management algorithms. Two of these algorithms are based on well-known scheduling policies: first come, first serve (in short, FCFS) and an approach based on the close page policy (in short, CPP). The other 2 management algorithms won the Memory Scheduling Championship: high performance memory access (in short, HPMA) and request density aware fair memory (in short, RDAF) (Balasubramonian, 2012).

Table 1 shows the list of parameters used for modelling different memory systems in the USIMM simulator (Chatterjee et al., 2012), where the first column refers to the name of the parameter in the simulator and the second column presents a description of the parameter. The first six parameters are related with architectural constraints, such as the number of channels, ranks and banks of the memory system. The rest of these parameters refers to latency constraints, such as row to column interval delay, column access and row cycle interval delay, among others.

7.2. Assessing the suitability of the knowledge base (RQ1)

The main objective of this first experiment is to check the suitability of the knowledge base, which uses MRs to represent the behaviour of memory systems. Initially, we assume that the scheduling algorithms and the simulator are correct.

In this experiment, the testing process consists in executing 25,000 different test cases, generated from combining 500 memory models and 50 benchmarks, over the original system. This process is carried out for the 4 memory management algorithms. Also, in those cases where the workload is executed in parallel, an additional simulation is performed. Table 2 depicts the results of this experiment, which shows the percentage of test cases that fulfil each MR using the 4 memory scheduling algorithms. These results show that the major part of the MRs is satisfied by all the test cases. In contrast, only MR_6 and MR_{10} are fulfilled by none of the test cases.

For those cases where a given MR is satisfied for all the test cases, we assume this MR correct. However, there are 2 possible scenarios for those cases where none of the test cases satisfy the MR: the MR is not correct or, on the contrary, the simulator has a limitation to execute some test cases.

In order to reject the assumption that the MR is not correct, we have carefully designed a few test cases that should fulfil the involved MRs. For instance, to check MR_6, we use $\omega = \omega' \cdot \omega'$, that is, the workload executed over m is the result of concatenating the workload ω' to itself. Since both memories m and m' are equal, the power required to execute ω over m must be greater than the power required to execute ω' over m'. Similarly, we have manually generated some tests to check the correctness of MR_{10}. However, we obtain different results than expected and, therefore, we conclude that the simulator has a limitation to represent certain scenarios.

Consequently, since MR_6 and MR_{10} are not able to evaluate the underlying behaviour of the memory system using USIMM, these MRs have been removed from the KB and not used in the following experiment.

7.3. Assessing the effectiveness of the expert system (RQ2)

In this section we evaluate the effectiveness of our proposed expert system for finding errors in memory management systems. In order to accomplish this analysis, we have used mutation testing to generate 5 different mutants from each memory management algorithm. These mutants reproduce typical errors committed by programmers while designing memory management policies. For this, different faults have been seeded in the main parts of the memory controller,like the scheduler and the read/write queue management (delaying operations, modifying the queue, etc). The mutations are summarised in Table 3. The algorithms used in this study are independent to each other and, therefore, their source code is different in all cases. Hence, it has not been possible to exactly create the same mutants for each algorithm and thus a collection of errors has been specifically adapted for each planner.

The testing process of this experiment uses the same test cases that were generated in Section 7.2. However, in this case, these tests are executed over the original system and the generated mutants. Overall, we required the execution of more than one million of simulations to accomplish this experiment.

Figs. 9, 10, 11 and 12 show the effectiveness of each MR to detect faults in each memory management algorithm. For the sake of clarity, the results of each experiment are depicted in two charts. The chart on the top provides the results in a compact form, showing the percentage of unsuccessful test cases (i.e., those that do not discover an error). The chart on the bottom shows a detailed view of each test case execution, which are numbered from 0 to 499. If the execution of a test case over a mutant satisfies the MR, the mutant is kept alive, which is represented in green. On the contrary, if the execution of a test case over a mutant does not fulfil the MR, a fault is detected and the mutant becomes killed, which is shown in red. In both charts, the x-axis shows the MRs involved in the testing process, where each MR is divided in 5 columns representing the generated mutants.

Table 4 shows the effectiveness of each MR for detecting faults in the tested memory system, that is, the percentage of test cases that do not fulfil the MR. To represent these results, we use the following notation: M_i^{sys} denotes the generated mutant i from the system sys; M_{Avg}^{sys} is the average effectiveness of each MR for checking all the mutants of the system sys (5 mutants are involved); $Total_{avg}$ is the average effectiveness of each MR for finding faults in all the generated mutants (20 mutants are involved). The first column of this table refers to the involved mutant(s) for

Table 2
Percentage of test cases that satisfy each MR.

Id	MR_1	MR_2	MR_3	MR_4	MR_5	MR_6	MR_7	MR_8	MR_9	MR_{10}
Cpp	100	100	100	100	100	0	100	100	100	0
Fcfs	100	100	100	100	100	0	100	100	100	0
Hpma	100	100	100	100	100	0	100	100	100	0
Rdaf	100	100	100	100	100	0	100	100	100	0

Table 3
Description of the generated mutants.

Id	Original Statement	Faulty statement	Description
M_1^{Cpp}	$if(wr\_ptr \rightarrow comm\_issuable)$	$if(!wr\_ptr \rightarrow comm\_issuable)$	Delaying a write operation
M_2^{Cpp}	$if(wr\_q\_len[ch] > HI)$	$if(wr\_q\_len[ch] < HI)$	Modifying the write queue
M_3^{Cpp}	$rd\_q\_len[ch] - -$	$rd\_q\_len[ch]$	Modifying the read queue
M_4^{Cpp}	$if(!drain\_wr[ch])$	$if(drain\_wr[ch])$	Swapping a read for a write
M_5^{Cpp}	$wr\_q\_head[ch]$	$wr\_q\_head[++ch]$	Swapping operation channel
M_1^{Fcfs}	$if(drain\_wr[ch] \wedge wr\_q\_len[ch]$	$if(drain\_wr[ch]\|\|wr\_q\_len[ch])$	Forcing a write operation
M_2^{Fcfs}	$if(!rd\_q\_len[ch])$	$if(rd\_q\_len[ch])$	Modifying the read queue
M_3^{Fcfs}	$if(wr\_q\_len[ch] > HI)$	$if(wr\_q\_len[ch] < HI)$	Modifying the write queue
M_4^{Fcfs}	$if(wr\_ptr \rightarrow comm\_issuable)$	$if(!wr\_ptr \rightarrow comm\_issuable)$	Delaying a write operation
M_5^{Fcfs}	$for(ch = 0; ch < NC; ch++)$	$for(ch = 1; ch < NC; ch++)$	Bypassing a channel
M_1^{Hpma}	$if(issue\_request(rdat\_ptr))$	$if(!issue\_request(rdat\_ptr))$	Delaying a read operation
M_2^{Hpma}	$if(rdat\_ptr \neq NULL)$	$if(rdat\_ptr == NULL)$	Delaying a read operation
M_3^{Hpma}	$if(drain\_wr\|\|issue\_wact)$	$if(drain\_wr \wedge issue\_wact)$	Modifying the serving policy
M_4^{Hpma}	$if(wdat\_ptr \neq NULL)$	$if(wdat\_ptr == NULL)$	Delaying a write operation
M_5^{Hpma}	$switch(sboard[ch].state)$	$switch(sboard[ch].state++)$	Changing the controller state
M_1^{Rdaf}	$if(wr\_ptr \rightarrow comm\_issuable)$	$if(!wr\_ptr \rightarrow comm\_issuable)$	Delaying a write operation
M_2^{Rdaf}	$if(r\_ptr \rightarrow comm\_issuable)$	$if(!rd\_ptr \rightarrow comm\_issuable)$	Delaying a read operation
M_3^{Rdaf}	$if(wr\_q\_len[ch] > HI)$	$if(wr\_q\_len[ch] < HI)$	Modifying the write queue
M_4^{Rdaf}	$is\_TFW(ch, rank, cycle))$	$is\_TFW(ch, ++rank, cycle)$	Bypassing a rank
M_5^{Rdaf}	$state[ch][rank][bank].next$	$state[ch][rank][bank].next++$	Changing the controller state

Table 4
Effectiveness (in %) of each MR for detecting faults in memory scheduling systems.

Id	MR_1	MR_2	MR_3	MR_4	MR_5	MR_7	MR_8	MR_9	C_{13}	C_{45}	C_{47}
M_1^{Cpp}	99.8	100.0	98.2	98.0	100.0	97.2	92.2	100.0	100.0	98.2	100.0
M_2^{Cpp}	7.8	3.4	0.2	0.2	0.2	97.2	92.2	0.0	9.8	0.2	100.0
M_3^{Cpp}	7.8	100.0	0.2	0.2	0.2	97.6	92.2	0.0	9.8	0.2	100.0
M_4^{Cpp}	59.2	17.8	98.2	98.0	98.2	96.8	92.2	98.0	59.39	98.2	100.0
M_5^{Cpp}	99.8	100.0	98.2	98.0	98.2	96.2	92.2	100.0	100.0	98.2	100.0
M_{Avg}^{Cpp}	54.8	64.2	59.0	58.8	59.0	97.0	92.2	59.6	55.8	59.0	100.0
M_1^{Fcfs}	7.8	100.0	0.2	0.2	0.2	97.6	92.2	0.0	9.8	0.2	100.0
M_2^{Fcfs}	7.8	100.0	0.2	0.2	0.2	97.39	92.2	0.0	9.8	0.2	100.0
M_3^{Fcfs}	7.8	100.0	0.2	0.2	0.2	97.6	92.2	0.0	9.8	0.2	100.0
M_4^{Fcfs}	99.8	100.0	98.2	98.0	98.2	97.6	92.2	100.0	100.0	98.2	100.0
M_5^{Fcfs}	100.0	7.8	98.2	98.2	98.2	95.4	92.2	3.4	100.0	98.2	100.0
M_{Avg}^{Fcfs}	44.64	81.56	39.4	39.35	39.4	97.12	92.2	20.68	45.87	39.4	100.0
M_1^{Hpma}	99.8	97.2	98.2	95.20	98.2	94.2	93.2	95.8	100.0	95.4	100.0
M_2^{Hpma}	99.8	97.2	98.2	95.20	98.2	94.0	93.2	97.2	100.0	95.4	100.0
M_3^{Hpma}	99.6	97.0	98.2	95.20	98.2	94.0	93.2	95.8	99.8	95.4	100.0
M_4^{Hpma}	99.8	97.2	98.2	95.20	98.2	94.8	93.2	95.8	100.0	95.4	100.0
M_5^{Hpma}	99.8	97.2	98.2	95.20	98.2	93.8	93.2	93.8	100.0	95.4	100.0
M_{Avg}^{Hpma}	99.75	97.15	98.2	95.20	98.2	94.15	93.2	95.67	99.96	95.4	100.0
M_1^{Rdaf}	59.8	99.2	98.2	98.2	98.2	97.0	92.2	100.0	100.0	98.2	99.8
M_2^{Rdaf}	100.0	99.8	98.2	98.2	98.2	96.6	92.2	100.0	100.0	98.2	100.0
M_3^{Rdaf}	7.8	100.0	0.2	0.2	0.2	97.6	92.2	0.0	9.8	0.2	100.0
M_4^{Rdaf}	100.0	39.0	98.2	98.2	98.2	97.79	41.0	99.4	100.0	98.2	100.0
M_5^{Rdaf}	99.4	98.0	98.2	98.2	98.2	96.6	91.39	100.0	100.0	98.2	100.0
M_{Avg}^{Rdaf}	73.4	87.2	78.6	78.6	78.6	97.12	81.79	79.88	81.96	78.6	99.96
$Total_{avg}$	68.14	82.52	68.8	67.98	68.8	96.34	89.84	63.95	70.90	68.1	99.99

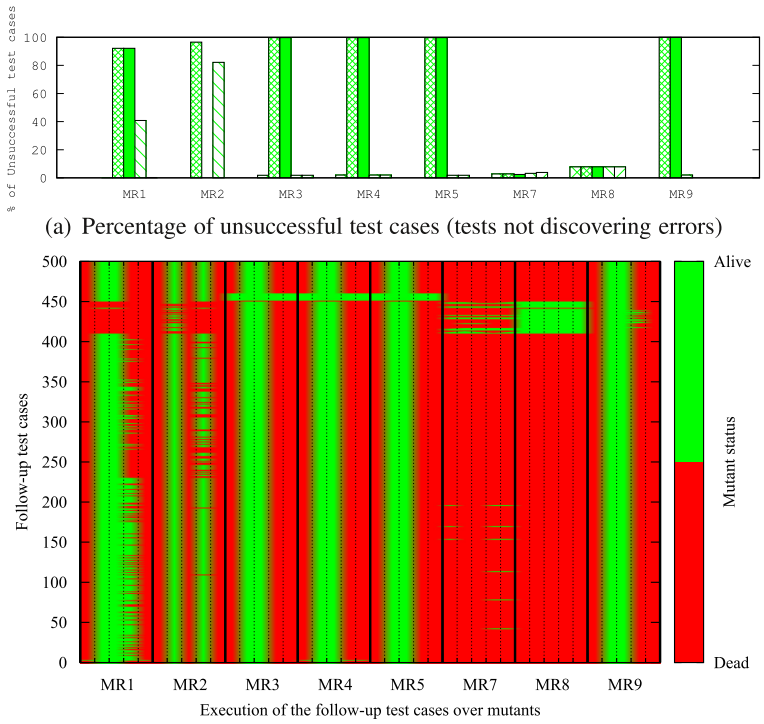


Fig. 9. Effectiveness of the proposed MRs for checking Close Page Policy.

calculating the effectiveness value. The next 8 columns represent the effectiveness of each MR to detect faults.

In general, the proposed expert system detects all the generated mutants, which represent faulty memory systems. However, each MR provides a different effectiveness, which depends on the memory management system under test. Let us remark that a low percentage of unsuccessful test cases in the MRs (top chart in Figs. 9, 10, 11 and 12) represents a high effectiveness to detect faults.

Fig. 9 shows the results for finding faults in the CPP algorithm. In this case, the most effective relations are MR_7 and MR_8, which have detected faults in the major part of the follow-up test cases. This is depicted in Fig. 9.a, where these MRs show a low percentage of unsuccessful test cases, achieving an effectiveness of 97% and 92.2%, respectively. The rest of the MRs provide a lower effectiveness, which ranges from 54.8% to 64.2%.This fact is reflected in the columns that contain both green and red lines (see Fig. 9.b). It is important to note that MR_9 is not able to detect mutants 2 and 3.

Fig. 10 shows the results obtained for checking the FCFS algorithm. While MR_2, MR_7 and MR_8 achieve an effectiveness of 81.56%, 97.12% and 92.2%, respectively, the rest of the MRs provides a significantly lower effectiveness, ranging from 20.68% to 44.64%. Similar to the previous analysed algorithm, MR_9 does not detect mutants 1, 2 and 3.

Fig. 11 shows the results for checking the HPMA algorithm. In this case, only a reduced number of test cases executed over the mutants have satisfied the MRs, which are represented with green stripes. Consequently, these MRs achieve, in average, an effectiveness of 96.44% for detecting all the mutants in this system.

Fig. 12 depicts the results for checking the RDAF algorithm. In this case, the most effective relations are MR_2, MR_7 and MR_8 achieving an effectiveness of 87.2%, 97.12% and 81.79%, respectively. However, in general, all the MRs provide acceptable results, achieving an effectiveness that ranges from 73.4% to 97.12%. In this algorithm, mutant 3 seems hard to kill, it is not detected by MR_9 and it is barely detected by MR_1, MR_3, MR_4 and MR_5.

It is important to mention that the MRs that provide the lowest average effectiveness values for detecting faults, have difficulties to kill mutants that have been created by modifying statements based on the read/write queue. In order to alleviate this drop in the average effectiveness, we have generated 3 new MRs by composing different MRs from Section 5. This way, C_{13} refers to the composition of relations MR_1 and MR_3, C_{45} refers to the compositions of MR_4 and MR_5, and C_{47} refers to the composition of MR_4 and MR_7. The idea is to complement each MR for detecting faults in a greater number of test cases than using a single MR. The testing process has been repeated by using the new generated MRs (see the last three columns of Table 4). The results obtained for C_{13} and C_{47} have achieved a better effectiveness than each single MR, reaching an average effectiveness of 70.9% and 99.99%, respectively. In this case, these new MRs provide an accurate model to represent the behaviour of the system under test and, therefore, the results obtained are promising. However, C_{26} achieves a lower effectiveness than each single MR, that is, MR_2 and MR_6. Thus, we observe that, in this case, it is not practical to merge these MRs because

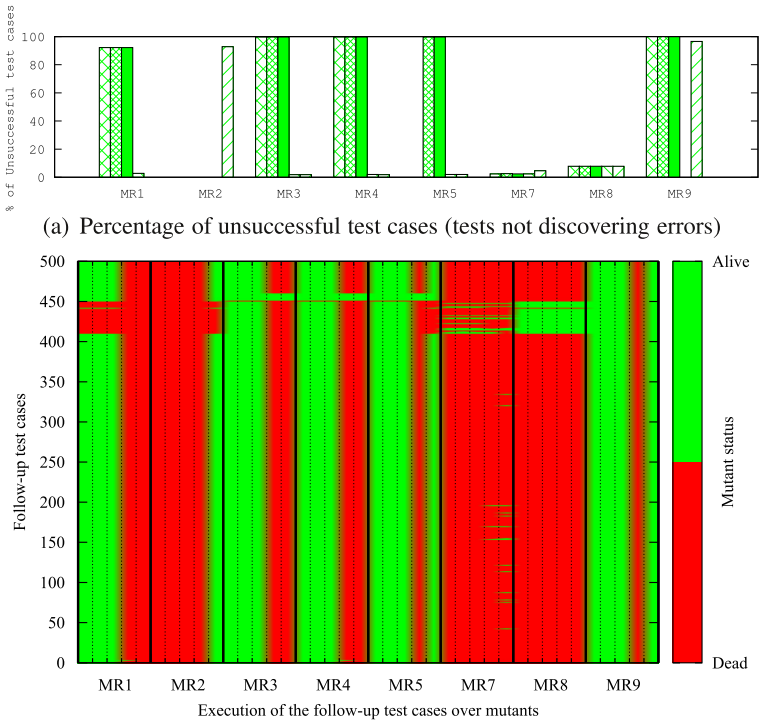


Fig. 10. Effectiveness of the proposed MRs for checking First Come, First Serve.

the conditions inside the relations (MR_i, MR_o) are not compatible for combination to accurately model the system under test.

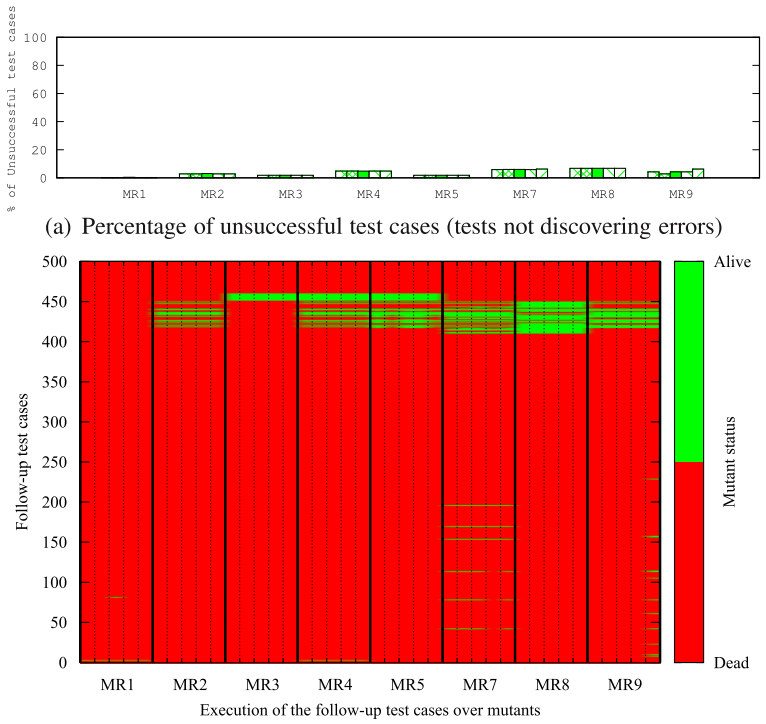
7.4. Comparison with current standard methods for testing memory systems (RQ3)

This section presents a study that compares our proposed expert system with a current standard method for testing memory systems. To the best of our knowledge, expert systems have not been applied to test memories in the past. Thus, since the main objective of this work is to automatically test memory systems using a wide range of configurations, we think the most appropriate method to be used in this study is random testing, which despite its simplicity, is widely used in computer systems and applications (Arcuri, Iqbal, & Briand, 2012; Ciupa et al., 2011; Hassan et al., 2015; Rixner et al., 2000).

We are particularly interested in investigating the effectiveness of random testing to detect faults in memory systems. Hence, we have carried out an experiment where 1000 test cases are randomly generated for testing the four different memory scheduling algorithms analysed in previous sections (CPP, FCFS, HPMA and RDAF). Again, for each algorithm, the same five faults were artificially injected. Similarly, other 1000 test cases have been generated, using our proposed system, to detect the same faults. The idea is to compare the effectiveness to detect the injected faults of our system against the one reached by random testing.

Table 5shows the results of this study, where each column - namely $Fault_1$ to $Fault_5$ - represents the different injected faults in the memory schedulers. Each row represents the system under test (in short, SUT) and consists of two different values, where *valid* shows the percentage of test cases that are successfully executed and *detect* is the percentage of *valid* test cases that detects the fault. For instance, in order to test the CPP scheduler, random testing generates 31 valid test cases – from the 1000 test cases generated in total – where only 58.06% of these test cases (18 in total) are able to detect $Fault_1$. In this experiment, our system (in short, ES) executes the testing process using C_{47}, which is the best combination obtained in the previous study (see Section 7.3). Random testing (in short, Rnd) does not apply constraints to generate the test cases.

In general, our proposed system provides better results than random testing. These results show that, when random testing is applied to generate the test cases, there is a considerable number of them that are not valid. In some cases, a test case contains a wrong hardware configuration of a memory system that cannot be simulated like, just to name a few, wrong number of channels, wrong size and non-compatible frequencies and, therefore, only valid test cases can be executed to test the memory. Additionally, we observe that, in the major part of the cases, our system clearly outperforms random testing. There are few cases where random testing provides the same results – in percentage of test cases – to detect the fault. However, it is important to remark that our proposed system generates quality test cases, in the sense that almost all the generated cases are valid, for testing memory systems, which increases the overall performance of the testing process. Also, using combined MRs clearly provides the best results



(a) Percentage of unsuccessful test cases (tests not discovering errors)

(b) Detailed results of the testing process

Fig. 11. Effectiveness of the proposed MRs for checking High Performance Memory Access.

Table 5
Effectiveness (in %) of random testing vs. our proposed system using composed MRs.

SUT		$Fault_1$		$Fault_2$		$Fault_3$		$Fault_4$		$Fault_5$	
	Rnd	ES	Rnd	ES	Rnd	ES	Rnd	ES	Rnd	ES	
CPP	Valid	3.10	99.20	2.90	99.60	3.00	99.20	3.10	99.01	3.20	99.41
	Detect	58.06	100	89.65	100	90	100	48.38	100	41.93	100
FCFS	Valid	3.10	99.10	2.90	99.80	2.90	99.65	3.00	99.90	3.10	99.10
	Detect	90.32	100	100	100	89.65	100	60.00	100	100	100
HPMA	Valid	3.00	99.6	2.90	99.5	3.10	99.40	3.10	99.5	3.10	99.6
	Detect	60.00	100	27.58	100	58.06	100	60.00	100	19.35	100
RDAF	Valid	3.00	99.6	2.90	99.5	3.10	99.4	3.10	99.5	3.10	99.6
	Detect	20.00	99.8	37.93	100	90.32	100	67.74	100	27.58	100

for testing memory systems achieving, in the worst case scenario, 99.8% of effectiveness to detect faults.

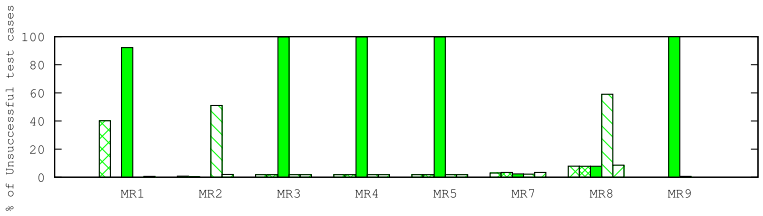
Next, we compare the effort required to carry out the testing process. Using random testing requires relatively few effort to generate and execute the test cases. Basically, the tester only has to identify the input parameters that must be randomly generated to create the test suite. However, random testing requires the manual creation of an oracle, or manually checking the provided outputs, which is a very time consuming task.

Similar to random testing, our system only requires effort for preparing the test cases if the MRs do not exist yet. In that case, the method to generate the test suite must be adapted for the involved MRs. However, in contrast to random testing, our expert system uses the constraints defined in the MRs to automatically generate the test cases and, therefore, the major part of the test cases are valid memory configurations. Our method solves the oracle problem, because once the test cases are executed, our system is able to automatically check the provided outputs using the defined MRs. Hence, checking whether a test is successful is automatic in our case.

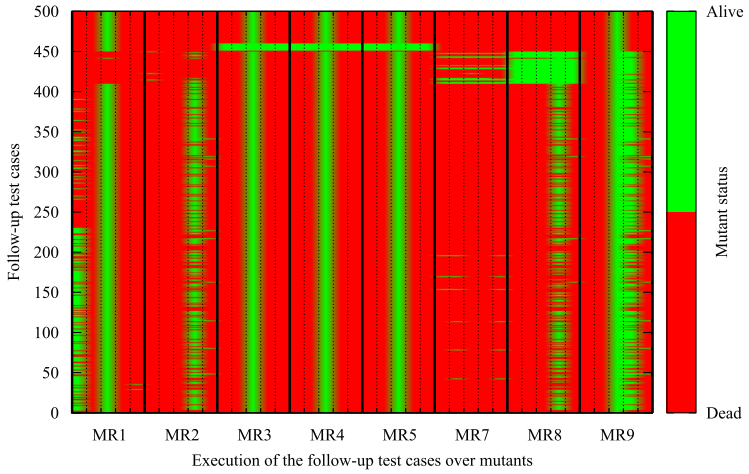
In conclusion, random testing provides a lower effectiveness to test the memories than our proposed system and requires a considerable effort to check the provided results. Hence, we think that our proposed system is a valuable contribution for the research community, not only for automatically generating quality test cases, but to alleviate the oracle problem, eliminating the effort of the tester to check the provided output.

7.5. Discussion of the results

In this section, we answer the research questions using the results obtained from the previous experiments.



(a) Percentage of unsuccessful test cases (tests not discovering errors)



(b) Detailed results of the testing process

Fig. 12. Effectiveness of the proposed MRs for checking Request Density Aware Fair Memory.

In order to answer **RQ1:** "*Are the designed MRs suitable to be used as rules in the knowledge base?*", we use the results described in Section 7.2. These results were obtained by using the USIMM simulator and 4 different memory management algorithms, which we assumed correct. Table 2 summarises these results, which clearly show 2 different behaviours for each MR, that is, all the tests satisfy the MR or, by the contrary, none of the test cases satisfy the MR. It is interesting the fact that exactly the same results are obtained for the 4 memory management algorithms. However, as we described in Section 7.2, these results are mainly produced by the shortcomings of the USIMM simulator (Chatterjee et al., 2012) and, consequently, we decided to remove these MRs from the KB. Since the 100% of the test cases satisfied the rest of the MRs, we assume that these are suitable to be used as rules in the knowledge base.

Next, we answer the research question **RQ2:** "*How effective is the proposed expert system to catch errors in faulty memory systems?*", by using the results obtained in Section 7.3, which are summarised in Table 4. In general, those rules representing critical aspects of the system provide promising results. Relations MR_2, MR_7 and MR_8 achieve by far the best results, reaching an average effectiveness of 82.52%, 96.34% and 89.84%, respectively. Those MRs focusing on general aspects of the system provide acceptable results. In this case, relations MR_1, MR_3, MR_4 and MR_5 exceed 63% of average effectiveness in all these cases. However, since MR_9 is not able to detect several mutants in 3 of the previously analysed algorithms, we can state that this a useless MR to detect those kind of faults. Also, we can observe a correlation between the complexity of the analysed scheduling algorithms and the effectiveness of each MR. The two first algorithms, FCFS and CPP, are less complex than HPMA and RDAF. However, the results show that the analysed MRs provide the best effectiveness in those algorithms that have a high complexity. Since HPMA and RDAF contain sensible parts, a small change in those algorithms generates an unexpected behaviour that is easily detected by the MRs. On the contrary, basic algorithms contain less parts of code that are sensible to produce unexpected behaviour and, consequently, the seeded errors are more difficult to detect. Additionally, we have identified some MRs that detect similar errors. For instance, MR_1, MR_3, MR_4 and MR_5 provide promising results for detecting errors focusing on read/write delays. However, since these MRs focus on general aspects of the system, these do not provide specific information to locate the error. Hence, we conclude that the proposed expert system is effective to detect anomalies in the memory management system, providing better results in complex algorithms.

To answer **RQ3:** "*How suitable is the proposed expert system to test memory systems compared with standard methods?*", we have carried out an experiment for comparing our proposed expert system with random testing to test faulty memory systems. We observe that random testing requires few effort to generate and execute test cases. However, it suffers from the oracle problem, because the tester has to manually check the provided outputs, which is a tedious and error prone task. This is alleviated by our approach, which allows to automatically execute the testing process. The results obtained using our approach are better than the ones obtained by random testing. First, we show that our approach

is more efficient to create quality test cases. Second, we obtain the best effectiveness to detect faults when the system combines different MRs to execute the testing process, which clearly outperforms the effectiveness provided by random testing. Hence, the answer to this question is that our proposed system is fairly suitable to test memory systems because, in contrast to other standard techniques, the testing process is automatically executed obtaining better effectiveness.

After a careful analysis, we can conclude that the results provided by the experiments carried out in the empirical study are promising. Our system is not only able to detect faults in the memory system, but also to show those bugs that are more complicated to detect, which is calculated using the effectiveness of each MR. It is important to note that an accurate design of the MRs is key for detecting errors. In this work MR_2, MR_7 and MR_8 are able to detect the major part of the faults. We have also investigated the impact of combining different MRs for detecting faults, showing that the right combination of MRs increases the system effectiveness. However, using wrong combinations provides worse results than using MRs individually. Finally, we found that the expert has a significant impact on the system performance, that is, providing an accurate design of MRs and properly combining MRs is directly reflected in the overall system effectiveness.

The main strengths of our proposed frameworks focuses on the two problems previously described: the oracle problem and the reliable test set problem. On the one hand, we show that the proposed expert system is able to automatically test a wide range of memory configurations using large test suites. Thus, the tester is able to execute the testing process without manually checking the result of each test. On the other hand, the novel design of the proposed expert system, which includes in its core a data-base, a simulator and a collection of MRs, allows automatically generating appropriate test suites. We show that the generated tests are able to identify bugs in different memory scheduling algorithms. However, our proposed system also has some limitations. We think that the most relevant weakness of our proposed system lies in the design of the MRs, which must be manually designed by the tester. These MRs are integrated in the core of the expert system and are used both to generate the test cases and to automatically check the output provided by the tests execution. Hence, providing accurate and appropriate MRs is key for the proper functioning of the system. A challenge to the community is the automatic discovery of MRs from the observation of execution traces using e.g., machine learning techniques (Kanewala, Bieman, & Ben-Hur, 2016). A second limitation is that the workload used to test the memory system must be representative. That is, using small and non-realistic workloads might not be suitable for accurately testing the memory system. In this work we alleviate this issue using a wide spectrum of workloads inspired by the PARSEC benchmarks (Bienia et al., 2008).

7.6. Threats to validity

In this section, we discuss the threats to validity of our empirical study.

7.6.1. Internal threats

Internal validity is concerned with whether our findings (based on the obtained results from the empirical study) truly represent a cause-and-effect relationship. Thus, the internal validity of our study lies in the implementation of our experiments.

The design of the KB is based on the experience of two experts. We are aware that the ability of the expert system to detect errors highly depends on the selection of metamorphic properties and, therefore, the results may have varied if different MRs were used to build the KB. However, the use of domain-specific properties, like the ones used to design our proposed catalogue of MRs, should reveal a high percentage of failures (Xie et al., 2009).

We have implemented the MRs in Java and used USIMM to simulate a wide spectrum of scenarios to obtain the results. These results are used to check if the MRs are fulfilled, or not. We have conducted code inspection and run different tests by hand to assure the correctness of these implementations. Moreover, the source code has been checked by different individuals. Our evaluation of the MRs is based on the randomly generated inputs, that is, the source test cases. Similarly, the follow-up test cases have been generated by using random values and the corresponding constrains to assure the relation between the source test case and the generated one is fulfilled.

The chosen mutation operators could be another threat to internal validity. Different operators and hand-seeded faults may produce different mutants. However, we carefully designed the mutation operators by investigating, from different sources, common faults produced by programmers, including works in the current literature, repositories of different simulators, “whats new” logs and mailing lists.

Other issues might arise due to the simulator used. This might have errors that can affect our findings. The USIMM simulator, which represents the behaviour of different scenarios of memory systems to execute the tests, has been widely used by the research community. Moreover, this simulator has been in the Memory Scheduling Championship (Chatterjee et al., 2012). We mitigate this threat with the experiment described in Section 7.2, where a broad range of test cases, involving 500 different memory models, were executed and checked over our proposed MRs.

7.6.2. External threats

External validity is concerned with the extent to which the results of a study can be generalised.

We have used 500 different memory configurations and 50 different workloads, inspired by the PARSEC benchmarks. Although we believe that these models represent a broad range of memory configurations, there is no guarantee that the obtained results and the achieved improvements of effectiveness of the MRs are the same for other scenarios.

We have chosen four memory management algorithms to conduct our empirical study. The purpose of choosing these algorithms is to analyse planners with different degrees of complexity. While two of these algorithms are relatively simple, FCFS and CPP, the complexity of the other two algorithms, HPMA and RDAF, is significantly higher. However, more experiments would be needed to fully warrant the generalisation of our proposal.

7.6.3. Construct threats

Construct validity is concerned with whether the used measures are representative or not.

We measured the quality of the expert system based on its fault-detection effectiveness, which is also widely used in the community.

Defects in the simulator or in our proposed expert system could be a threat to construct validity. We controlled this threat by executing a wide spectrum of test cases, using four different memory management algorithms, over the USIMM simulator. After this experiment, we removed 2 MRs from the KB because we detected a limitation in the simulator, which does not properly represent the properties reflected in the discarded MRs. Hence, we check that the MRs were properly designed and that our implementation worked correctly.

8. Conclusions

In this paper we have presented an expert system for detecting faults in memory systems using simulation and metamorphic testing. For this, we have built the KB using MRs, which address critical aspects of memory systems, such as functional, performance and energy consumption. Additionally, we use the methodology of metamorphic testing to automatically generate test cases, which are simulated in a memory simulator.

To measure the effectiveness of our proposed expert system, we performed an experimental study based on mutation testing. In general, the proposed expert system achieves promising results, detecting the major part of the injected faults. The MRs focused on specific aspects of the system achieve better results than those focused on general aspects. In addition, the composition of MRs outperforms the results of each single MR. A comparison with random testing shows that our approach requires less effort to validate the test results (the MRs solve the oracle problem) while achieving higher effectiveness.

We can conclude that, during the testing process, the role of the expert is of vital importance. First, the expert is in charge of designing the MRs. Second, her decisions to choose those MRs to be combined – and included into the KB – have a direct impact of the final obtained results.

Since the proposed system presents a novel design, which integrates simulation and metamorphic testing in its core, different lines for future work have been opened. First, we plan to develop a new language for designing constraints in memory systems. Thus, new MRs could be easily created by the expert and integrated into the expert system, which will increase the spectrum of tested memory configurations. Using this language, the expert system will be able to automatically check the MRs designed by the expert. The expert might provide wrong MRs that do not accurately represent the behaviour of the memory system and, therefore, an automatic checking would provide a significant value to the system effectiveness. Another line of research is to use machine learning methods to analyse the FDB of performed simulations, and infer possible MRs, in the style of Kanewala et al. (2016).

Additionally, in order to provide a solid contribution to the scientific community, we also plan to create a public repository of MRs where researchers could share and use these relations into their own systems. Hence, using a proper DSL, we will allow not only to design new MRs for testing memory systems, but to share these MRs in the public repository. Finally, due to the widely adoption of cloud services by the research community, the proposed service could be deployed in the cloud as a service, which allows researchers accessing to the proposed system through the Internet and without installing additional software into their computers.

Author Contribution

Pablo C. Cañizares and Alberto Núñez conceived the idea of this work. Pablo C. Cañizares developed the Expert System. Alberto Núñez and Juan de Lara designed the testing process and the theoretical framework and Pablo C. Cañizares collected data and performed the computations.

All authors discussed the results, have been involved in writing the manuscript and participated in the required revisions. All authors agree on the order in which their names will be listed in the manuscript.

Credit authorship contribution statement

Pablo C. Cañizares: Methodology, Data curation, Investigation. **Alberto Núñez:** Methodology. **Juan de Lara:** Methodology, Supervision.

Acknowledgments

This work was supported by the Spanish Ministerio de Economía, Industria y Competitividad, Gobierno de España/FEDER (grant numbers DArDOS, TIN2015-65845-C3-1-R and FAME, RTI2018-093608-B-C31) and the Comunidad de Madrid project FORTE under Grant S2018/TCS-4314. The first author is also supported by the Universidad Complutense de Madrid - Santander Universidades grant (CT17/17-CT18/17).

Conflict of interest

There are no conflicts of interest to declare.

References

Abts, D., Jerger, N. D. E., Kim, J., Gibson, D., & Lipasti, M. H. (2009). Achieving Predictable Performance Through Better Memory Controller Placement in Many–core CMPs. In *Proceedings of the 36th international symposium on computer architecture* (pp. 451–461).

Arcuri, A., Iqbal, M. Z., & Briand, L. (2012). Random testing: theoretical results and practical implications. *IEEE Transactions on Software Engineering, 38*(2), 258–277.

Awasthi, M., Nellans, D. W., Sudan, K., Balasubramonian, R., & Davis, A. (2010). Handling the problems and opportunities posed by multiple on-chip memory controllers. In *Proceedings of the 19th international conference on parallel architecture and compilation techniques* (pp. 319–330). ACM.

Balasubramonian, R. (2012). Memory scheduling championship results. http://www.cs.utah.edu/~rajeev/jwac12/results_table.html. Accessed July, 2017.

Barr, M. (2017). EmSA: Products, Consulting & Training for Embedded Systems - Software Based Memory Testing. http://www.esacademy.com/en/library/technical-articles-and-documents/miscellaneous/software-based-memory-testing.html. [Online; Accessed on Dec 15, 2017].

Bennett, J., & Hollander, C. (1981). DART: An expert system for computer fault diagnosis. In *Proceedings of the 7th international joint conference on Artificial intelligence-Volume 2* (pp. 843–845). Morgan Kaufmann Publishers Inc..

Bienia, C., Kumar, S., Singh, J. P., & Li, K. (2008). The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on parallel architectures and compilation techniques* (pp. 72–81). ACM.

Cañizares, P. C., Núñez, A., Núñez, M., & Pardo, J. J. (2015). A methodology for designing energy-aware systems for computational science. In *Proceedings of the 15th international conference on computational science: 51* (pp. 2804–2808). Elsevier.

Chatterjee, N., Balasubramonian, R., Shevgoor, M., Pugsley, S. H., Udipi, A. N., Shafiee, A., Sudan, K., Awasthi, M., & Chishti, Z. (2012). USIMM: the Utah SImulated Memory Module A Simulation Infrastructure for the JWAC Memory Scheduling Championship.

Chen, T. Y., Cheung, S. C., & Yiu, S. M. (1998). Metamorphic testing: a new approach for generating next test cases. *Technical Report*. HKUST-CS98-01.

Ciupa, I., Pretschner, A., Oriol, M., Leitner, A., & Meyer, B. (2011). On the number and nature of faults found by random testing. *Software Testing, Verification and Reliability, 21*(1), 3–28.

Clarke, E. M., Grumberg, O., & Peled, D. A. (2001). *Model checking*. MIT Press.

C.M. Herrero-Jiménez (2012). An expert system for the identification of environmental impact based on a geographic information system. *Expert Systems with Applications, 39*(8), 6672–6682.

Council, J. E. D. E. (2017). JEDEC DDR4 SDRAM Standard, 2012. https://www.jedec.org/standards-documents/docs/jesd79-4a. Accessed: 2017-07-30.

de Dinechin, B. D., van Amstel, D., Poulhies, M., & Lager, G. (2014). Time-critical computing on a single-chip massively parallel processor. In *Proceedings of the design, automation & test in europe conference & exhibition* (pp. 1–6).

Ding, J., Zhang, D., & Hu, X.-H. (2016). An application of metamorphic testing for testing scientific software. In *Proceedings of the 1st international workshop on metamorphic testing* (pp. 37–43). ACM.

Dreibelbis, J., Barth, J., Kalter, H., & Kho, R. (1998). Processor-based built-in self-test for embedded DRAM. *IEEE Journal of Solid-State Circuits, 33*(11), 1731–1740.

Gepner, P., & Kowalik, M. F. (2006). Multi-core processors: New way to achieve high system performance. In *Proceedings of the 5th international conference on parallel computing in electrical engineering* (pp. 9–13). IEEE.

Ghasempour, M., Jaleel, A., Garside, J. D., & Luján, M. (2016). DReAM: Dynamic re-arrangement of address mapping to improve the performance of drams. In *Proceedings of the 2nd international symposium on memory systems* (pp. 362–373). ACM.

de Goor, A. J. V., & Smit, B. (1994a). Automating the verification of memory tests. In *Proceedings of the 12th ieee vlsi test symposium* (pp. 312–318). IEEE.

de Goor, A. J. V., & Smit, B. (1994b). Generating march tests automatically. In *Proceedings of the international test conference* (pp. 870–878). IEEE.

Gundu, A., Sreekumar, G., Shafiee, A., Pugsley, S. H., Jain, H., Balasubramonian, R., & Tiwari, M. (2014). Memory bandwidth reservation in the cloud to avoid information leakage in the memory controller. In *Proceedings of the 3rd workshop on hardware and architectural support for security and privacy* (pp. 11:1–11:5).

Hardee, K. C., Chapman, D. B., & Pineda, J. (1991). Dynamic random access memory. US Patent.

Hassan, M., & Patel, H. (2017). MCXplore: Automating the validation process of DRAM memory controller designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Hassan, M., Patel, H., & Pellizzoni, R. (2015). A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *Proceedings of the 21st ieee real-time and embedded technology and applications symposium* (pp. 307–316). IEEE.

Hierons, R. M., Merayo, M. G., & Núñez, M. (2010). Mutation testing. In *Encyclopedia of software engineering* (pp. 594–602). Wiley-Interscience.

Huang, C.-T., Huang, J.-R., Wu, C.-F., Wu, C.-W., & Chang, T.-Y. (1999). A programmable BIST core for embedded DRAM. *IEEE Design & Test of Computers, 16*(1), 59–70.

Hussain, A., S. J. Lee, M. S. Choi, & Brikci, F. (2015). An expert system for acoustic diagnosis of power circuit breakers and on-load tap changers. *Expert Systems with Applications, 42*(24), 9426–9433.

Ipek, E., Mutlu, O., Martínez, J. F., & Caruana, R. (2008). Self-optimizing memory controllers: A reinforcement learning approach. In *2008 international symposium on computer architecture* (pp. 39–50).

J. Bautista-Valhondo, & R. Alfaro-Pozo (2018). An expert system to minimize operational costs in mixed-model sequencing problems with activity factor. *Expert Systems with Applications, 104*, 185–201.

Jacob, B., Ng, S., & Wang, D. (2007). *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc..

Jeong, M. K., Yoon, D. H., & Erez, M. (2012). DrSim: A platform for flexible DRAM system research.

Jiang, M., Chen, T. Y., Kuo, F.-C., & Ding, Z. (2013). Testing central processing unit scheduling algorithms using metamorphic testing. In *Proceedings of the 4th ieee international conference on software engineering and service science* (pp. 530–536). IEEE.

Kanewala, U., Bieman, J. M., & Ben-Hur, A. (2016). Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels. *Softw. Test., Verif. Reliab., 26*(3), 245–269.

Karpovsky, M. G., van de Goor, A. J., & Yarmolik, V. N. (1995). Pseudo-exhaustive word-oriented DRAM testing. In *Proceedings of the 1995 european conference on design and test* (p. 126). IEEE Computer Society.

Kayed, M. O., Abdelsalam, M., & Guindi, R. (2014). A novel approach for SVA generation of DDR memory protocols based on TDML. In *Proceedings of the 15th international microprocessor test and verification workshop* (pp. 61–66). IEEE.

Khalifa, K., & Salah, K. (2015). Implementation and verification of a generic universal memory controller based on UVM. In *Proceedings of the 10th international conference on design & technology of integrated systems in nanoscale era* (pp. 1–2). IEEE.

Kim, Y. (2017). Ramulator: A Fast and Extensible DRAM Simulator. https://github.com/CMU-SAFARI/ramulator. [Online; Latest commit on Dec 11, 2016. Accessed on Dec 15, 2017].

Kim, Y., Han, D., Mutlu, O., & Harchol-Balter, M. (2010). ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proceedings of the 16th international conference on high-performance computer architecture* (pp. 1–12).

Kim, Y., Yang, W., & Mutlu, O. (2016). Ramulator: A Fast and extensible DRAM simulator. *IEEE Computer Architecture Letters, 15*(1), 45–49.

Kumar, P. (2017). USIMM: the Utah SImulated Memory Module. https://github.com/pranith/usimm. [Online; Latest commit on Oct 22, 2014. Accessed on Dec 15, 2017].

Li, Y., Akesson, B., Lampka, K., & Goossens, K. (2016). Modeling and verification of dynamic command scheduling for real-time memory controllers. In *Proceedings of the 2016 ieee real-time and embedded technology and applications symposium* (pp. 1–12). IEEE.

Liberado, E., ao, F. M., oes, M. S., W. A. de Souza, & Pomilio, J. (2015). Novel expert system for defining power quality compensators. *Expert Systems with Applications, 42*(7), 3562–3570.

Lin, W.-F., Reinhardt, S. K., & Burger, D. (2001). Reducing DRAM latencies with an integrated memory hierarchy design. In *Proceedings of the 7th international symposium on high-performance computer architecture* (pp. 301–312). IEEE.

Liu, H., Kuo, F.-C., Towey, D., & Chen, T. Y. (2014). How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering, 40*(1), 4–22.

Liu, H., Liu, X., & Chen, T. Y. (2012). A new method for constructing metamorphic relations. In *Proceedings of the 12th international conference on quality software* (pp. 59–68).

Mahapatra, N. R., & Venkatrao, B. (1999). The processor-memory bottleneck: problems and solutions. *Crossroads, 5*(3es), 2.

Miyano, S., Sato, K., & Numata, K. (1999). Universal test interface for embedded–DRAM testing. *IEEE design & test of computers, 16*(1), 53–58.

Modgil, A., Nitin, & KumarSehgal, V. (2015). Understanding and Analyzing the Impact of Memory Controller's Scheduling Policies on DRAM's Energy and Performance. *Procedia Computer Science, 70*(Supplement C), 399–406. Proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems

Mukundan, J., Hunter, H., hyoun Kim, K., Stuecheli, J., & Martínez, J. F. (2013). Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems. *SIGARCH Computer Architecture News, 41*(3), 48–59.

Natarajan, C., Christenson, B., & Briggs, F. (2004). A study of performance impact of memory controller features in multi-processor server environment. In *Proceedings of the 3rd workshop on memory performance issues: In conjunction with the 31st international symposium on computer architecture*. In *WMPI '04* (pp. 80–87).

Núñez, A., & Hierons, R. M. (2015). A methodology for validating cloud models using metamorphic testing. *Annals of Telecommunications, 70*(3–4), 127–135.

Poremba, M. (2017). NVMain - An Architectural Level Main Memory Simulator for Emerging Non-Volatile Memories. https://bitbucket.org/mrp5060/nvmain. [Online; Latest commit on Oct 18, 2017. Accessed on Dec 15, 2017].

Pundir, A. K., & Sharma, O. (2017). CHECKERMARC: A Modified Novel memory-Testing approach for bit-Oriented SRAM. *International Journal of Applied Engineering Research, 12*(12), 3023–3028.

Rixner, S., Dally, W. J., Kapasi, U. J., Mattson, P., & Owens, J. D. (2000). Memory access scheduling. In *Proceedings of the 27th international symposium on computer architecture (isca'00)* (pp. 128–138). ACM.

Rogers, B. M., Krishna, A., Bell, G. B., Vu, K., Jiang, X., & Solihin, Y. (2009). Scaling the bandwidth wall: challenges in and avenues for CMP scaling. *SIGARCH Computer Architecture News, 37*(3), 371–382. doi:10.1145/1555815.1555801.

Rosenfeld, P. (2017). DRAMSim2: A cycle accurate DRAM simulator. https://github.com/umd-memsys/DRAMSim2. [Online; Latest commit on Nov 8, 2014. Accessed on Dec 15, 2017].

Rosenfeld, P., Cooper-Balis, E., & Jacob, B. (2011). DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters, 10*(1), 16–19.

Sahoo, D., & Satpathy, M. (2016). MSimDRAM: Formal model driven development of a dram simulator. In *Proceedings of the 29th international conference on embedded systems and vlsi design* (pp. 597–598). IEEE.

Segura, S., Fraser, G., Sanchez, A. B., & Ruiz-Cortés, A. (2016). A survey on metamorphic testing. *IEEE Transactions on Software Engineering, 42*(9), 805–824.

Shafiee, A., Gundu, A., Shevgoor, M., Balasubramonian, R., & Tiwari, M. (2015). Avoiding information leakage in the memory controller with fixed service policies. In *Proceedings of the 48th international symposium on microarchitecture* (pp. 89–101).

Sherwood, T., Perelman, E., Hamerly, G., Sair, S., & Calder, B. (2003). Discovering and exploiting program phases. *IEEE Micro, 23*(6), 84–93.

Standard, J. (2012). DDR3 SDRAM Standard. *JESD79-3, JEDEC*.

Subramanian, L., Lee, D., Seshadri, V., Rastogi, H., & Mutlu, O. (2016). Bliss: balancing performance, fairness and complexity in memory access scheduling. *IEEE Transactions on Parallel and Distributed Systems, 27*(10), 3071–3087.

Sudan, K., Chatterjee, N., Nellans, D., Awasthi, M., Balasubramonian, R., & Davis, A. (2010). Micro-pages: Increasing DRAM Efficiency with Locality-aware Data Placement. In *Proceedings of the 15th international conference on architectural support for programming languages and operating systems*. In *ASPLOS XV* (pp. 219–230). ACM.

Weyuker, E. J. (1982). On testing non-testable programs. *The Computer Journal, 25*(4), 465–470.

Wu, C.-F., Huang, C.-T., Cheng, K.-L., & Wu, C.-W. (2000). Simulation-based test algorithm generation for random access memories. In *Proceedings of the 18th ieee vlsi test symposium* (pp. 291–296). IEEE.

Wulf, W. A., & McKee, S. A. (1995). Hitting the memory wall: Implications of the obvious. *SIGARCH Computer Architecture News, 23*(1), 20–24. doi:10.1145/216585.216588.

Xie, X., Ho, J., Murphy, C., Kaiser, G., Xu, B., & Chen, T. Y. (2009). Application of metamorphic testing to supervised classifiers. In *Ninth international conference on quality software* (pp. 135–144).

Yang, C.-C., Li, J.-F., Yu, Y.-C., Wu, K.-T., Lo, C.-Y., Chen, C.-H., … Chou, Y.-F. (2015). A hybrid built-in self-test scheme for DRAMs. In *Proceedings of the 2015 international symposium on vlsi design, automation and test* (pp. 1–4). IEEE.

7.2 MT-EA4Cloud: A Methodology for testing and optimising energy-aware cloud systems

7.2	
Publication data	
Authors:	Pablo C. Cañizares, Alberto Núñez, Juan de Lara and Luis Llana.
Title:	MT-EA4Cloud: A Methodology for testing and optimising energy-aware cloud systems
Publication:	Journal of Systems and Software
Pub. Type:	Journal
Year:	Under 3^{rd} round of revision
DOI/URL:	–
Pages:	–
Category:	Computer Science, Software Engineering
Quartile:	Q1
Ranking:	26/107
Impact factor:	2.559
Contribution	
Summary:	This proposal presents a formal approach to check the correctness – from an energy-aware point of view – of cloud systems and optimise their energy consumption. To make the checking of energy consumption practical, MT-EA4Cloud combines metamorphic testing, evolutionary algorithms and simulation. Metamorphic testing allows to formally model the underlying cloud infrastructure in the form of metamorphic relations. We use metamorphic testing to alleviate both the reliable test set problem, generating appropriate test suites focused on the features reflected in the metamorphic relations, and the oracle problem, using the metamorphic relations to check the generated results automatically. MT-EA4Cloud uses evolutionary algorithms to efficiently guide the search for optimising the energetic consumption of cloud systems, which can be calculated using different cloud simulators
Technique:	Metamorphic Testing
Secondary techniques:	Mutation Testing, Simulation, Evolutionary Algorithms, Formal Modelling, AI

MT-EA4Cloud: A Methodology for testing and optimising energy-aware cloud systems

Pablo C. Cañizares[1] , Alberto Núñez[1], Juan de Lara[2] and Luis Llana[1]

[1]*Dept. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Spain*

e-mail: pablocc@ucm.es, alberto.nunez@pdi.ucm.es, llana@ucm.es

[2]*Dept. Ingeniería Informática, Universidad Autónoma de Madrid, Spain*

e-mail: juan.delara@uam.es

Abstract

Currently, using conventional techniques for checking and optimising the energy consumption in cloud systems is unpractical, due to the massive computational resources required. An appropriate test suite focusing on the parts of the cloud to be tested must be efficiently synthesised and executed, while the correctness of the test results must be checked. Additionally, alternative cloud configurations that optimise the energetic consumption of the cloud must be generated and analysed accordingly, which is challenging.

To solve these issues we present MT-EA4Cloud, a formal approach to check the correctness – from an energy-aware point of view – of cloud systems and optimise their energy consumption. To make the checking of energy consumption practical, MT-EA4Cloud combines metamorphic testing, evolutionary algorithms and simulation. Metamorphic testing allows to formally model the underlying cloud infrastructure in the form of metamorphic relations. We use metamorphic testing to alleviate both the reliable test set problem, generating appropriate test suites focused on the features reflected in the metamorphic relations, and the oracle problem, using the metamorphic relations to check the generated results automatically. MT-EA4Cloud uses evolutionary algorithms to efficiently guide the search for optimising the energetic consumption of cloud systems, which can be calculated using different cloud simulators.

Keywords: Cloud modelling, Metamorphic testing, Simulation, Evolutionary algorithms, Energy-aware systems

1. Introduction

Cloud computing platforms are currently increasing their role to perform large-scale computational analysis [1, 2]. The ever-growing amount of online data requires the use of a large number of computational resources – like CPUs and storage devices – for its efficient processing. However, the speedup obtained by exploiting the parallelism offered by these resources requires an extremely high energetic cost [3]. As an example, according to the most recent survey (in November 2018) of the fastest 500 computers in the world, the top supercomputer Summit reaches a performance of 200,794.9 Teraflop/s with 2,397,824 cores [4]. This system requires 9,783 kW of power, a cost of 1,858€ per hour if we assume a cost of 0.19€ per kW.

In the last decade, IT companies have invested much effort to reduce energy costs. This has led to developing new approaches for improving the efficiency of energy consumption on cloud computing systems [5, 6]. However, testing these systems requires a high level of expertise, not only to design appropriate test suites but also to find a compromise between the overall performance and the efficiency to detect faults. Unfortunately, testing these approaches is costly, challenging and, in some cases, unfeasible.

Although testing is currently the most widely used technique to validate the correctness of computing systems, it presents some drawbacks when it is applied to cloud systems. First, the underlying architecture of the system under test is needed for checking its correctness. Moreover, the access to the cloud settings is usually restricted (i.e. allocation policy of virtual machines, network topology) which hampers the testing process. This is so as testing involves executing a large test suite over the system under test, which requires dedicated access to the system during a long period of time. Finally, providing an appropriate test suite (including the oracle) for determining the correctness of a system is complex, and particularly challenging if we target energy consumption. Hence, conventional testing techniques are not suitable to face the problem of checking the correctness and optimising energy consumption in cloud systems.

In order to alleviate these issues, we propose a formal methodology, called `MT-EA4Cloud`, to efficiently check the correctness of energy-aware cloud systems, and automatically propose improvements to their designs. `MT-EA4Cloud` is founded on a novel combination of metamorphic testing [7, 8], evolutionary algorithms [9] and simulation. `MT-EA4Cloud` requires the user to provide the cloud model under study, and hence it focuses on the testing and optimisation phases. This way, metamorphic testing is applied in the testing phase, EAs are applied to the optimisation phase and the energy consumption of each cloud model is calculated using different simulators.

Metamorphic testing (MT) is a testing technique that alleviates the oracle problem [7, 8] and the reliable test set problem [10]. Basically, MT models the properties of the system under test as *metamorphic relations* (MRs). The essential idea is that instead of checking the output o_1 produced when testing with one input x_1, we test with a second (follow-up) input x_2, observing output o_2, and check that o_1 and o_2 are related as specified by the MRs. `MT-EA4Cloud` uses MRs – which formally model the underlying cloud infrastructure – to automatically generate test cases focusing on the features reflected in these relations. Moreover, the MRs are applied to automatically check the correctness of the results provided by the execution of the test cases.

Evolutionary Algorithms (EAs) are inspired by the principle of natural selection and genetics [9]. The idea consists of simulating the evolution of different individuals, each one representing a potential solution to the target problem, which are evaluated using a fitness function. In this work, each individual represents a cloud configuration and the fitness function is designed by using a catalogue of MRs modelling the underlying behaviour of cloud systems regarding energy consumption. The main motivation to apply EAs for optimising the energy consumption of cloud systems is that EAs focus on an adaptive global search in the space of possible solutions and provide near-optimal solutions to complex and hard optimisation problems, where the execution time represents a significant constraint. Thus, `MT-EA4Cloud` applies EAs to efficiently generate alternative cloud configurations (individuals) to optimise the energetic consumption.

During the last years, simulation has been widely adopted by the research community as a cost-effective technique to model and analyse cloud computing systems [11, 12, 13]. Simulation presents several advantages in this domain: the access to the underlying architecture of the system under test is not required; experiments can be easily reproduced and parallelised, which significantly reduces the total execution time; and a wide spectrum of cloud configurations can be easily generated.

Overall, our work makes the following four contributions:

- Providing a methodology for deciding the most appropriate simulator to model and simulate cloud infrastructures. In general, each simulator focuses on modelling and simulating a specific part of the cloud like, just to name a few, the storage system, virtual machine (VM) migration, resource provisioning and energy consumption. The high number of existent cloud simulators makes it difficult to select an appropriate simulator for a specific purpose. In this work, we use MT to alleviate this issue. We accurately model the cloud in the form of MRs, which represent the underlying behaviour of the cloud. Thus, our methodology allows measuring the adequacy of each simulator for simulating cloud systems. This process is carried out semi-automatically, in the sense that the test cases are automatically generated, executed and checked, but the MRs must be manually designed by the expert.

- Proposing a methodology for optimising the energy consumption of cloud infrastructures. Broadly speaking, the main difficulty in optimising cloud systems lies in the high complexity of constructing an accurate model of the system. In order to check and optimise the energy consumption of cloud systems, we propose a novel approach that combines MT and EAs. The idea is to use our provided EA to evolve cloud systems efficiently. This is achieved by using MRs, which guide the search, for finding an optimised cloud. This way, each new offspring of individuals (clouds) are generated using the constraints defined in the MRs. Thus, the proposed EA not only reduces the search space but improves the overall efficiency.

- Providing an automatic method to execute the testing process. Optimising the energy consumption of cloud systems requires to generate an appropriate test suite for determining the correctness of the system under test (known as the reliable test set problem [10]) and to decide if the outputs of a test suite are correct (known as the oracle problem [14]). We face these challenges by combining MT and simulation. Our methodology allows to automatically generate quality test cases using the provided MRs and therefore the created test suite focuses on testing the features reflected in the MRs. Additionally, the outputs of the simulations are automatically checked using MRs. Thus, we provide an automatic method for optimising cloud systems without the intervention of a (human) oracle.

- Evaluating our proposed methodology to study its applicability to optimise cloud systems from an energy consumption point of view. We present a thorough study to optimise three different cloud systems using seven well-known cloud simulators. First, we use our methodology to check the adequacy of each simulator to analyse the cloud systems. Second, we use our novel approach that combines MT and EAs for optimising the energy consumption of the clouds. Finally, we show that our method outperforms an approach that uses MT and random testing for generating test cases.

The rest of the paper is structured as follows. Section 2 presents an overview of energy-aware cloud computing and introduces the main concepts of MT. Section 3 analyses related works, stressing the novelty of our methodology with respect to them. A detailed description of our proposed methodology is presented in Section 4. In Section 5 we present a catalogue of MRs to analyse energy consumption in cloud systems. Section 6 shows our EA to optimise the energetic consumption in cloud systems, and Section 7 presents tool support. Section 8 describes

a thorough experimental study using our proposed methodology with different simulation tools. The threats to validity of our experiments are discussed in Section 9. Finally, Section 10 finishes with the conclusions and prospects for future work.

2. Background

In this section we provide an introduction to some of the main concepts used in this work: energy-aware cloud computing, and metamorphic testing.

2.1. Energy-awareness in cloud computing

Currently, the energy consumption in cloud systems is gaining attention as one of the main important concerns in this field. In general, a high energy cost can be considered as a threat since it decreases the Return of Investment (ROI) and increases the Total Cost of Ownership (TCO) of the cloud infrastructures. The economic impact of running these systems cannot be neglected by its users, but moreover, the whole world is affected. The exponential increase of energy consumption has produced significant changes in the global environment, which has a negative environmental impact and in the well-being of the inhabitants of the world. On the one hand, the data centres – which in general are overprovisioned – constantly operate under their maximum capacity [15]. On the other, the developers of the applications that are executed on them generally do not take the energy consumption into consideration [16, 17]. In some cases, a single data centre is able to produce 170 million metric tons of carbon per year [18]. The carbon emissions of data centres worldwide are expected to reach 670 million metric tons by 2020 [19]. In particular, the average concentration of carbon dioxide in the atmosphere has increased from 280 ppm in the year 1750, to more than 400 ppm in 2018 [20].

Several factors motivate the interest in detecting and reducing the main causes of energy consumption, such as carbon footprint reduction [21], savings in IT electricity bills [22], and increase of the life time of some devices [23]. *Green computing* [24], or sustainable computing, has become the focus attention of initiatives such as Green Grid [25], a global consortium dedicated to advancing energy efficiency in data centres and business computing ecosystems. As demonstrated by the successful emergence of the Green500 list [26], which provides a ranking of the most energy-efficient supercomputers in the world, energy consumption has become as significant as performance.

2.2. Metamorphic testing

Conventional testing methods require checking whether the output(s) returned by the system under test are the expected ones. Schematically, let S be a system, I the input domain, $\mathcal{X}$ a test selection strategy and $\mathcal{T} = \{t_1, t_2, \ldots, t_n\} \subseteq I$ the set of tests generated using $\mathcal{X}$. When these tests are sequentially applied to the system S we obtain a sequence of outputs $S(t_1)$, $S(t_2)$, …, $S(t_n)$. Therefore, if we have an oracle, called f, to predict the expected output of S when exercised with any test in $\mathcal{T}$, then we find an error if there exists $t_i \in \mathcal{T}$ such that $S(t_i) \neq f(t_i)$.

In general, testing faces two fundamental problems. First, the *oracle problem* [14], which refers to the availability of a mechanism to distinguish between the correct behaviour and potentially incorrect behaviours of the system under test. Unfortunately, in some situations – like testing cloud systems – an oracle is not available or its application is computationally too expensive and alternative approaches must be used. The second issue is the *reliable test set problem* [10], which consists in providing an appropriate test suite for determining the correctness of a system.

Since it is normally not feasible to execute all possible test cases over the system under test, a subset needs to be selected. However, selecting the most optimal subset is challenging.

Metamorphic testing, unlike the major part of the testing techniques, can be applied for both test case generation and test result verification, making it suitable to face both fundamental problems of testing [27, 28, 29]. The main difference between traditional testing techniques and MT lies in the comparison of the obtained outputs. Hence, while traditional techniques compare the output of each individual test case with the one obtained from the oracle, MT checks the relation between multiple test inputs and their outputs.

MT uses expected properties of the target system, relating multiple test inputs with the corresponding outputs obtained from the system under test. These properties are formulated as metamorphic relations (MRs). An MR is a property of the analysed system that involves multiple inputs and their outputs. We represent an MR as a formulae $\mathsf{i}(MR) \implies \mathsf{o}(MR)$, where $\mathsf{i}(MR)$ refers to the relation between the source test case and the follow-up test case, and $\mathsf{o}(MR)$ refers to the relation that must be fulfilled by the outputs obtained from the source test case and the follow-up test case. Generally, follow-up test cases are automatically generated by applying modifications over a source test case.

Let us illustrate this method with an example. Consider the problem of testing an implementation of the trigonometric *sine* function. Any implementation of this function will be an approximation, where the process for checking the correctness of an expected output for a given input can be complex and error-prone. However, we can define MRs encoding our knowledge of the domain, like the fact that $sine(x) = -sine(-x)$. Then, we provide a test input t, say 0.3, and call the function with it. Next, we calculate a follow-up test f that permits exercising the MR ($f = -0.3$). Finally, we check whether the two obtained outputs are related as expected by the MR ($sine(t) = -sine(f)$). If the output relation does not hold, a failure has been detected.

3. State of the art

In this section, we review works related to each technique used in our methodology: energy-aware cloud systems (Section 3.1), MT for cloud systems (Section 3.2), EAs in cloud design and operation (Section 3.3), and simulation of cloud systems (Section 3.4).

3.1. Energy-aware cloud systems

Current studies have shown that an idle data centre consumes around 70% of the power with respect to the same servers running at maximum CPU capacity [30]. Therefore, for energy efficiency reasons, it is necessary to devise techniques to hibernate idle nodes for reducing the overall consumption [31]. Next, we review some of them.

Sayadnavard and collaborators propose an approach that takes into account the reliability of each physical machine (PM) to reduce the number of active PMs at the same time [32]. Then, a Markov chain model is designed to analyse the reliability of PMs, which are prioritised based on the CPU usage and the reliability status. The effectiveness of this work has been evaluated by performing an experimental study using the CloudSim toolkit.

Mohhamad and collaborators present a VM placement method based on the balance-based cultural algorithm for virtual machine placement (BCAVMP) to decrease the energy consumption in cloud data centres [33]. The proposed algorithm provides a novel fitness function to estimate VM allocation solutions. Haghighi and collaborators propose a virtualisation technique for resource management [34]. For this, the authors suggest a hybrid technique based on k-means

for mapping task and dynamic consolidation and a micro-genetic algorithm. The proposed technique provides a good trade-off between reducing the energy consumption and the quality of service of data centres. There exist several methodologies for energy-efficient computing and networking, which are focused on reducing the energy consumption of security protocols and frameworks [35]. Samy and collaborators propose secure energy-aware provisioning of cloud computing resources on consolidated and virtualised platforms [36]. This work is based on a dynamic round-robin provisioning mechanism, which powers down the subsystems of a host that are not required by the requested VMs. The experimental evaluation of these works has been conducted using the CloudSim simulator.

Kharchenko and collaborators propose a method to examine the energy efficiency of computational tasks in hybrid clouds, taking into account data privacy and security aspects [37]. For this purpose, the authors examine a high computational demanding application by using mathematical models based on Markovian chains and queuing theory.

Overall, these approaches focus on saving energy in cloud systems by applying run-time techniques, like VM migration, server consolidation and VM placement. However, the goal of `MT-EA4Cloud` is to optimise the design of the cloud infrastructure for energy efficiency.

3.2. Mutation Testing for cloud systems

During the last years, MT [8, 7] has been successfully applied as an effective approach to alleviating the oracle problem to a wide variety of domains including, among others, web services [38] and embedded systems [39]. Remarkably, MT was able to detect new faults [40, 41] in three out of seven programs in the Siemens suite [42], which has been studied in major software testing research projects for 20 years. Similarly, Le and collaborators [43] discovered over one hundred faults in two popular C compilers (GCC and LLVM) using MT. Chan and collaborators [44] used MT to check both the functional behaviour and the energy consumption of wireless sensor networks. The authors of this work present two MRs for detecting failures, which are focused on data consolidation and equivalent consumption of sensor nodes that are close in proximity. Although MRs focusing on sensor networks must constantly be dealing with energy-saving issues, these cannot be applied to cloud computing systems. First, sensor networks are provided with limited batteries that considerably restrict the operations for computation and transferring information. On the contrary, the cloud uses computing and storage nodes that are provided with a constant power supply. Second, the cloud focuses on virtualising the hardware of computing nodes, allowing several users to share the resources of the same machine, which cannot be applied to sensor networks due to computing power limitations. Finally, the cloud is deployed using a predefined network topology, while the topology of the sensor networks may be built at run-time, allowing variations when the nodes have low battery.

Although there is currently work in the literature that combines MT with simulation techniques, to the best of our knowledge, MT has not been appropriately applied to check the correctness of energy consumption in cloud systems. Núñez and Hierons [45] combine the iCanCloud simulator with MT to detect unexpected behaviours when simulating cloud provisioning and usage. Although that contribution provides interesting ideas for checking the correctness of cloud systems, it also presents some limitations. That approach only provides a single MR for checking the correctness of the energy consumption. Moreover, a reduced number of test cases were applied during the testing process. Murphy and collaborators [46] present an approach to systematically test simulation software, specifically focusing on the domain of health care, with the aim of discovering defects in the implementation. Ding and collaborators [47] investigate the

effectiveness of MT to test a Monte Carlo modelling program for heterogeneous media. Additionally, they evaluate the adequacy of testing coverage criteria to measure the quality of the MT process, to guide the creation of MRs, in order to generate test inputs and investigate the exceptions found. Chen and collaborators propose the application of MT to check the conformance between network protocols and network simulators [48], and apply MT to discover faults in open queuing network models [49].

3.3. Evolutionary Algorithms in cloud design and operation

EAs have been successfully applied to solve real-world problems in a wide spectrum of fields, like swarm robotics, hardware design and fault tolerance and reconfigurability [50]. In particular, several works applying EAs to cloud systems can be found in the literature as well. Keshanchi and collaborators proposed an improved genetic algorithm for static task scheduling for cloud environments [51], for assigning subtasks to processors. A profile-based approach was developed by Vasudevan and collaborators for energy-efficient application assignment to VMs with consideration of resource utilization [52]. The approach is based on a Repairing Genetic Algorithm (RGA) to solve the large-scale optimisation problem. Xiao and collaborators [53] proposed a novel algorithm based on evolutionary game theory that successfully addresses the challenges faced by the dynamic placement of VMs. In this work, the authors demonstrate that the energetic consumption of a cloud is reduced by dynamically adjusting VMs placement. A dynamic task scheduling algorithm that uses an Integer Linear Programming (ILP) model focusing on minimising the energy consumption in a cloud data centre has been developed by Ibrahim and collaborators [54]. The authors of that work also propose an Adaptive Genetic Algorithm (AGA) to reflect the dynamic nature of the cloud environment, which provides a near-optimal scheduling solution that minimises energy consumption.

These works use EAs to optimise cloud systems but are focused on managing and scheduling tasks. Our approach uses MRs – previously designed by an expert – to adapt the search of an optimised cloud configuration using an EA. In particular, we provide the design of cloud infrastructure, including its hardware architecture, to optimise the energetic consumption of the cloud. To the best of our knowledge, there are only a few works in the literature combining EAs and MT. Segura and collaborators presented a proof of concept to automate the detection of performance bugs by combining MT and search-based techniques [55]. Rounds and Kanewala identified 17 MRs for testing a GA and show, through MT, that these relations are more effective at finding defects than traditional unit tests based on known outputs [56]. Arora and Bassi [57] show that using GAs increases the efficiency of MT to detect faults in software. However, these works focus on applying MT to check and test evolutionary algorithms, while our approach focuses on the evaluation of the energy consumption of cloud systems.

The current literature reports different techniques based on EAs, such as Genetic Algorithms (GAs), Genetic Programming (GP), Ant Colony Optimization (ACO), Cat Swarm Optimization (CSO), Particle Swarm Optimization (PSO) and Honey-Bees Mating Optimization (HBMO), among others. Despite the diversity, all the EA variants are based on a common working scheme, where the performance and accuracy obtained strongly depend on the type and the complexity of the problem [58, 59]. Thus, the task of selecting an EA technique to solve a computational problem is vital to successfully design a valid solution.

In our proposal, we need to model complex systems that require a high number of interrelated parameters and, therefore, we need flexibility, to generate a wide spectrum of cloud configurations (individuals), and high performance, to evaluate them efficiently. Techniques inspired

by ACO, CSO, PSO and HBMO are based on swarm intelligence, where the focus is on cooperation. In essence, swarm intelligence focuses on those systems containing many individuals that coordinate using decentralised control and self-organisation. Since our individuals must compete to obtain the best result (energy consumption), we discarded these approaches. On the contrary, GAs promote competition, where the individuals more adapted to the environment propagate their *genetic information* to the next generation of individuals. Thus, the best individuals of each generation have a higher probability of being selected for reproduction. GAs have been used in the past to solve problems related to the cloud [60, 61]. Moreover, we think that mutation and crossover techniques are suitable to face the problem of optimising energy consumption in cloud systems. For this, we propose a hybrid encoding – combining GAs and GPs – based on graphs and integer representation that eases the processing of the large structures that conform the cloud.

3.4. Simulation of cloud systems

The research community has developed a vast collection of tools for modelling and simulation of cloud systems. However, only a small subset of them focuses on analysing the energy consumption of the cloud [62]. This set includes, among others, CloudSim [11], DCSim [63], GreenCloud [64], SimGrid [13], iCanCloud [65, 12] and DISSECT-CF [66].

CloudSim is an extensible and open-source Java simulator, which enables modelling cloud computing systems and application provisioning environments. CloudSim is considered the *de facto* standard cloud simulation platform due to its capabilities for simulating cloud systems, such as VM allocation and provisioning, energy consumption, federated clouds and the possibility to model different types of clouds like public, private, hybrid and multi-cloud environments. One of the key features of CloudSim is the possibility to include new functionalities using extensions like *cloudSimStorage*, which supports modelling the energy consumption of the storage system [67].

DCSim, also known as *The Data Centre Simulator*, is a Java extensible simulation framework for simulating a data centre hosting. In essence, DCSim focuses on the IaaS layer for providing services to multiple tenants.

GreenCloud is an open-source tool for simulating data centres focusing on data communication and energy cost in cloud computing. GreenCloud provides a wide range of network and communication configurations for simulating data centres.

SimGrid is a tool for simulating algorithms and distributed applications in distributed computing platforms. The resources are modelled by their latency and service rate, and the topology is configurable by the users. Initially, SimGrid targeted grid environments. However, the current version of SimGrid supports a variety of cloud computing use cases including multi-purpose network representation, VM abstraction, live migration, VM support and storage.

iCanCloud is a simulation platform aimed to model and simulate cloud computing systems by providing different functionalities like resource provisioning. Additionally, the framework *E-mc*^{2} [12] can be used for analysing energy consumption. The main goal of iCanCloud is to predict the trade-offs between cost and performance of a given set of applications executed in specific hardware.

DISSECT-CF is a simulator targeted to evaluate the energy consumption of IaaSs. DISSECT-CF offers two major benefits: a unified resource-sharing model, and a complete IaaS stack simulation, which includes VM image repositories, storage and in-data-centre networking.

In general, the current approaches for modelling and simulating cloud systems are suitable to represent the behaviour of cloud architectures. However, each simulation tool focuses on a specific part of the cloud (e.g. storage system, VMs allocation policies, energy consumption)

and, unfortunately, there is no common solution that satisfies the entire research community. Moreover, these tools lack a formal approach to represent cloud systems, which makes difficult to automate the testing process. Our proposed approach focuses on alleviating these issues. First, using different simulation tools allows increasing the features of the cloud infrastructure that can be modelled and simulated. Second, combining MT and simulation allows to formally model the main features of the cloud and, therefore, automating the testing process. Finally, our methodology can propose cloud optimisations from the energy point of view, which is not supported by these tools.

4. Methodology

This section describes our proposed methodology, called `MT-EA4Cloud`, which combines MT, simulation and EAs to check the correctness of energy-aware cloud systems. The main steps of `MT-EA4Cloud` are depicted in Figure 1.

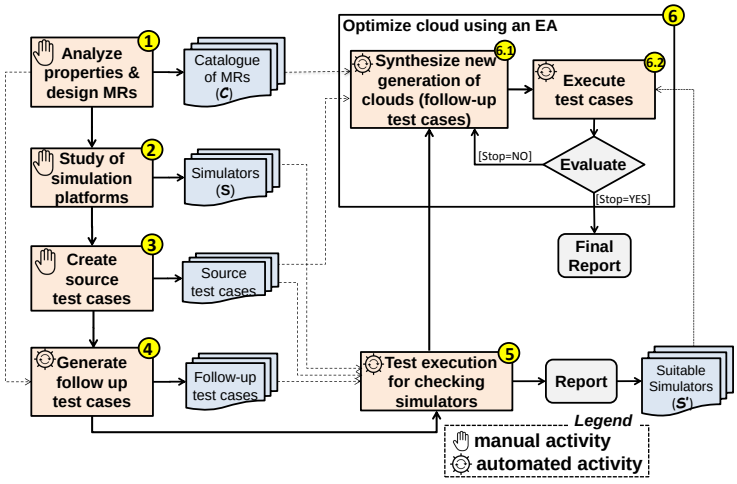


Figure 1: Scheme of the `MT-EA4Cloud` methodology.

Initially, the features having a relevant impact on the energy consumption must be carefully analysed, like the computing system, the storage system, network features and the workload to be processed, among others. Next, these features are used to design the MRs (label ① in the Figure). The idea is to provide a formal and accurate model – in the form of MRs – that represents the underlying behaviour of the cloud. The set containing the provided MRs, which we refer to as *catalogue*, is denoted by C and will be discussed in detail in Section 5.

Our methodology does not target a specific tool, moreover, it is desirable to use several simulators in the testing process. To that end, the tester must choose the simulators that offer capabilities to model and simulate the features formulated in C (label ②). In this step, the simulators are not executed, but their specifications are analysed to determine whether they can be used in the testing process. The chosen set of simulators in this step is denoted by $\mathcal{S}$.

In essence, we define a test case as a tuple (m, ω), where m refers to a cloud model and ω is the workload to be executed over m. The cloud model contains details about the underlying architecture of the system, while the workload represents the operations performed by the cloud. Similarly, a follow-up test case is denoted by (m', ω').

The execution of a workload ω over a cloud model m is carried out by simulation. We denote by $S(m, \omega)$ the result of the simulation – using the simulator S – for executing the workload ω over the cloud m. In Section 5 we formally define when a test case $t = (m, \omega)$ and a follow-up test case $f = (m', \omega')$ satisfy an MR in a simulation performed by simulator S, which is denoted by $(t, f, S) \models MR$.

Next, in step ③, the tester manually designs a reduced number of source test cases. This set is called $\mathcal{T}$. The main difference between a source test case and a follow-up test case lies in the way the test case is generated. While a source test case is manually designed, a follow-up test case is automatically generated by using a source test case and an MR. In step ④, we apply a procedure, described in Section 5.3, to generate a set $\mathcal{F}$ of follow-up test cases.

The goal of the next step (label ⑤) is two-fold: first, to analyse the adequacy of each MR; second, to investigate how appropriate is each simulator to represent the behaviour of cloud systems, focusing on energy consumption. In order to accomplish these objectives, the source test cases and the follow-up test cases are executed on the simulators chosen in step ②. We measure the adequacy of an $MR \in C$ by calculating the percentage of follow-up test cases f, generated from each source test case $t \in \mathcal{T}$ and executed using the simulator S, that fulfil $(t, f, S) \models MR$. The adequacy of a metamorphic relation MR using a simulator S and the test selection strategy $\mathcal{T}$, written $adq_{\mathcal{T}}(MR, S)$, is a number between 0 and 1 calculated as follows:

$$adq_{\mathcal{T}}(MR, S) = \frac{\sum_{t \in \mathcal{T}} |\{(t, f, S) \mid f \in followUp(t) \ \wedge \ (t, f, S) \models MR\}|}{\sum_{t \in \mathcal{T}} |\{(t, f, S) \mid f \in followUp(t)\}|} \quad (1)$$

where $followUp(t)$ is the set of generated follow-up test cases for t. Hence, `MT-EA4Cloud` can be applied to compare different simulators for representing the behaviour of cloud systems focusing on energy consumption.

Once all the tests are executed over the simulators, a report containing the adequacy of the MRs is generated. Next, the tester uses this report to create a new list of simulators, discarding those that do not appropriately represent the behaviour of cloud systems. As a result, a new list of simulators, denoted by $\mathcal{S}'$, is generated.

The quality of the cloud designs – focusing on energy consumption – that were created by the tester in step ③, is optimised in the last step (label ⑥). In order to accomplish this task, we use an EA, which is discussed in detail in Section 6. The EA evolves an initial population of cloud models generated from the source model, which is provided by the user. Such evolution involves applying different operations to create a new generation of clouds, until one of the individuals fulfils the stop criteria (e.g. its the energetic consumption has been reduced a 5%).

5. Metamorphic relations for modelling energy-aware cloud systems

In this section, we provide a catalogue of novel MRs that are specially designed to analyse the proper use of energy in cloud architectures. For this purpose, we first introduce some notation (Section 5.1), then we present the catalogue (Section 5.2) and finish by describing how to generate follow-up test cases using the MRs (Section 5.3).

5.1. Notation

Clouds. One of the main challenges in using MRs for accurately analysing complex systems [7], like clouds, lies in appropriately representing – with enough fidelity – the high number of inter-related parameters that determine the behaviour of the system under test, such as, in the case of cloud systems, data centres containing a large number of physical machines, communication networks, concurrent access of different users and virtualisation, among others. We use the parameters provided by each simulator to model and estimate these systems, like the network latency or the CPU speed. Hence, the level of accuracy obtained directly depends on how each simulator processes these parameters to simulate the underlying behaviour of each subsystem. In order to provide a flexible and accurate configuration of the cloud, we use the following notation:

- We represent a processor *cpu* as a pair (s, n) where $s \in \mathbb{N}$ is the speed of *cpu* measured in MFlops and $n \in \mathbb{N}$ is the number of cores.
- A hard disk *hd* is a tuple (s, r, w) where $s \in \mathbb{N}$ denotes the total size in GBytes and $r, w \in \mathbb{N}$ are the read and write bandwidth of the disk, measured in Mbps, respectively.
- The RAM memory *mem* is a tuple (s, r, w) where $s \in \mathbb{N}$ is the total size measured in MBytes and $r, w \in \mathbb{N}$ are the read and write bandwidth in MBps, respectively.
- A *node* is a tuple (cpu, hd, mem) where *cpu* is its processor, *hd* is its hard disk, and *mem* is its RAM memory.
- A *board node* is a tuple $(Id, nodes = \{n_j\}_{j \in J})$ where $Id \in \mathbb{N}$ is its identifier, and $nodes$ is the set of the actual nodes.
- A *rack* is a tuple $(Id, boards = \{b_i\}_{i \in I})$ where $Id \in \mathbb{N}$ is its identifier and $boards$ is a set of actual board nodes.
- A network connection *net* is a tuple (src, dst, bw, lat) where $src, dst \in \mathbb{N}$ are the identifiers of the source and destination racks of *net*, $bw \in \mathbb{N}$ is the bandwidth of *net* measured in MBps and $lat \in \mathbb{N}$ is the latency measured in μs.
- We define a *cloud model* as a connected non-directed graph $m = (RS, C)$ where RS is the set of racks and C is the set of network connections.

Given a board node b, we use $b.nodes$ for its set of nodes, and similarly for other tuples.

Cloud metrics. To measure the energy consumption of cloud systems, we define the following metrics:

- $\Delta(m_{cpu})$ denotes the overall CPU performance of the cloud model m. Formally

$$\Delta(m_{cpu}) = \sum_{r \in m.RS} \sum_{b_i \in r.boards} \sum_{n_j \in b_i.nodes} n_j.cpu.s \cdot n_j.cpu.n$$

- $\Delta(m_{IO})$ denotes the overall I/O performance of the cloud model m. Formally $\Delta(m_{IO}) = (r, w)$ where

$$r = \sum_{r \in m.RS} \sum_{b_i \in r.boards} \sum_{n_j \in b_i.nodes} n_j.hd.r$$

$$w = \sum_{r \in m.RS} \sum_{b_i \in r.boards} \sum_{n_j \in b_i.nodes} n_j.hd.w$$

- $\Delta(m_{NET})$ denotes the overall network performance of the cloud model m. Formally,

$$\Delta(m_{NET}) = \frac{1}{|C|} \cdot \sum_{c \in m.C} c.bw$$

- $|m|$ denotes the number of physical machines contained in the cloud model m. Formally,

$$|m| = \sum_{r \in m.RS} \sum_{b_i \in r.boards} |b_i.nodes|$$

- $|vm|$ denotes the number of virtual machines contained in the cloud model m.

Workload. A workload ω is a trace of operations to be executed on the cloud system, that is, requests of VMs to be deployed in physical machines, storage operations and computing operations. The trace elements are taken from OP, the set of basic operations, and so $\omega \in \mathsf{OP}^*$. We say that $\omega \subseteq \omega'$ if ω is a subtrace of ω', and define similarly a strict subtrace $\omega \subset \omega'$ and trace equality $\omega = \omega'$.

Test case. We define a test case as a tuple (m, ω), where m is a cloud model and ω is a workload that is executed over m. Similarly, a follow-up test case is denoted by (m', ω'). The execution of a workload ω over a cloud model m is performed using simulation. Thus, we denote by $S(m, \omega)$ the result of a simulation where the workload ω is executed over the cloud m. We are especially interested in energy consumption, and hence write $\Omega(S(m, \omega))$ to represent the overall energy consumption required to execute the workload ω over m, which is calculated using the simulator S. We can assume – without losing generality – that this is an integer number because power measures like $12.345W$ can be seen as $12345mW$ in a smaller power unit.

Metamorphic relation. Intuitively, an MR can be seen as a tuple (MR_i, MR_o), where MR_i is a relation over the source test case and a follow-up test case, and MR_o is a relation over the results obtained from the execution of these test cases. These test cases must fulfil the input relation MR_i. To capture this intuition, a *metamorphic* relation MR_S for m and ω using S, can be formally represented as a set of 4-tuples:

$$\begin{aligned} MR_S = \Big\{ \langle (m, \omega), (m', \omega'), S(m, \omega), S(m', \omega') \rangle \mid \\ MR_i((m, \omega), (m', \omega')) \Rightarrow MR_o(S(m, \omega), S(m', \omega')) \Big\} \end{aligned} \tag{2}$$

5.2. Catalogue of Metamorphic Relations

In order to accurately model the underlying cloud infrastructure, an expert with deep knowledge in cloud systems must define a catalogue of MRs. The catalogue we propose in this work is depicted in Figure 2. Next, we discuss its intuitive meaning, where the parameters not mentioned in the explanation of the rules remains the same.

MR_1: If the CPU of m has better performance than the CPU of m', and ω and ω' are equal, then the amount of energy required to execute ω over m should be less than or equal to the one required to execute ω' over m'.

MR_2: If the model m contains more machines than the model m', and ω and ω' are equal, then the ratio between the number of machines of m and m' should be greater than or equal to the ratio

$$
\begin{array}{ll}
\mathbf{MR_1} & \Delta(m_{cpu}) > \Delta(m'_{cpu}) \wedge \omega = \omega' \implies \Omega(S(m,\omega)) \leq \Omega(S(m',\omega')) \\
\mathbf{MR_2} & |m| \geq |m'| \wedge \omega = \omega' \implies \frac{|m|}{|m'|} \geq \frac{\Omega(S(m,\omega))}{\Omega(S(m',\omega'))} \\
\mathbf{MR_3} & \frac{|m|}{|m'|} \geq \frac{\Delta(m_{cpu})}{\Delta(m'_{cpu})} \wedge \omega = \omega' \implies \frac{|m|}{|m'|} \geq \frac{\Omega(S(m,\omega))}{\Omega(S(m',\omega'))} \\
\mathbf{MR_4} & \Delta(m_{IO}) > \Delta(m'_{IO}) \wedge \omega = \omega' \implies \Omega(S(m,\omega)) \leq \Omega(S(m',\omega')) \\
\mathbf{MR_5} & \Delta(m_{NET}) > \Delta(m'_{NET}) \wedge \omega = \omega' \implies \Omega(S(m,\omega)) \leq \Omega(S(m',\omega')) \\
\mathbf{MR_6} & \Delta(m_{RAM}) > \Delta(m'_{RAM}) \wedge \omega = \omega' \implies \Omega(S(m,\omega)) \leq \Omega(S(m',\omega')) \\
\mathbf{MR_7} & |m| = |m'| \wedge |vm| > |vm'| \wedge \omega = \omega' \implies \Omega(S(m,\omega)) \geq \Omega(S(m',\omega')) \\
\mathbf{MR_8} & m = m' \wedge \omega \subseteq \omega' \implies \Omega(S(m,\omega)) \leq \Omega(S(m',\omega'))
\end{array}
$$

Figure 2: Catalogue of MRs for checking the correctness of cloud simulators.

between the energy consumption required to execute ω over m and the one required to execute ω over m'.

MR_3: If the ratio between the number of machines of m and m' is greater than or equal to the ratio between the CPU performance of m and the CPU performance of m', and ω and ω' are equal, then the ratio between the energy consumption required to execute ω over m and the one required to execute ω' over m' should be less than or equal to the ratio between the number of machines of m and m'.

MR_4: If the I/O performance of m is better than the I/O performance of m', and ω and ω' are equal, then the energy consumption required to execute ω over m should be less than or equal to the one required to execute ω' over m'.

MR_5: If the network performance of m is better than the network performance of m', and ω and ω' are equal, then the energy consumption required to execute ω over m should be less than or equal to the one required to execute ω' over m'.

MR_6: If the RAM memory performance of m is better than the RAM memory performance of m', and ω and ω' are equal, then the energy consumption required to execute ω over m should be less than or equal to the one required to execute ω' over m'.

MR_7: If the number of machines used in m and m' is equal, the number of virtual machines deployed in m is greater than the number of virtual machines deployed in m', and ω and ω' are equal, then the energy consumption required to execute ω over m should be greater than or equal to the one required to execute ω' over m'.

MR_8: If m and m' are equal and the workload ω is a subtrace of ω', then the energy required to execute ω over m should be less than or equal to the one required to execute ω' over m'.

It is important to note that we use general purpose cloud simulators for calculating the energy consumption in cloud systems like, among others, cloudSim, simGrid, iCanCloud and Green Cloud. These simulators are not focused on modelling and simulating the cooling system. In general, these well-known cloud simulators are more focused on modelling the global cloud infrastructure (e.g. network, virtualization, storage), where the cooling system is not taken into account. However, there exist several approaches [68, 69] to model the cooling system using some of these cloud simulators. Unfortunately, the source code and the binary are not available. Consequently, the cooling system must be computed separately using a specific simulator for this purpose. For instance, DCWorms [70] models the cooling system of distributing systems. However, DCWorms has not been designed to represent the underlying behaviour of the cloud. Also, coolSim [71] provides models to manage and design the airflow in data centres.

5.3. Generation of follow-up test cases

MT can be used to alleviate the *reliable test set problem* [10]. First, the follow-up test cases are generated using a source test case as basis. Source test cases are manually created by the tester and, therefore, these test cases should be designed to test specific features of the system under test. Second, since generated tests should fulfil previously defined constraints in the form of MRs, these are focused on testing *sensible* parts of the system for providing relevant information.

Given a set of source test cases, we automatically generate the follow-up test cases, by copying the source test case and applying a slight modification in the replica. The key to automatically generate appropriate follow-up test cases is to calculate the cloud parameters that are most likely to be selected to perform the modifications. For this, we use the provided catalogue of MRs.

In our approach, a small number of source test cases must be provided by the tester. Additionally, the tester must select one or several MRs and the number of follow-up test cases to be generated. Since each generated test case must fulfil the selected MRs, we focus on those parameters that are used in the MRs (i.e. CPU parameters when MR_1 is involved). Next, we generate a random value within a specific range, to be assigned to the selected parameter. This way, we avoid generating follow-up test cases with unreal values like e.g. a communication network with a bandwidth of 1Mbps, or a disk drive with a capacity of 5 MB.

6. An Evolutionary Algorithm to optimise energy-aware cloud systems

Cloud computing systems usually consist of thousands of components and therefore, making random modifications on its underlying architecture, like CPU, memory and network, hampers the search of optimal models. In order to alleviate this issue, we propose an EA to explore complex search spaces efficiently. In particular, the EA will evolve the cloud systems using the catalogue of MRs to conduct the search towards the optimisation of the energetic consumption. For this purpose, it is necessary to provide a precisely defined cloud configuration with the main features of the system, as we defined in Section 5.1. The main steps of our proposed EA are depicted in Figure 3, and are explained next.

Initialisation. The algorithm requires as input a set of MRs modelling the underlying behaviour of the cloud, a cloud model and the maximum size of the population. Each individual in the population represents a cloud configuration. In this phase, the tester designs manually a cloud model, which is used as a seed to generate the initial population. Since the algorithm performs a guided search, each individual is synthesised by mutating the initial cloud according to the constraints reflected in the MRs (see the mutation step of the algorithm).

Evaluation. The individuals are evaluated using a fitness function, which provides a numeric value representing the quality of the candidate solution. This value is provided by a cloud simulator that calculates the energy consumption of each cloud model.

Encoding. In order to create the offspring, clouds are encoded in a way that facilitates its manipulation during the next stages of the algorithm. Clouds are complex systems made of a high quantity of connected components. Thus, it is necessary to design an encoding that appropriately represents all the required features and minimises dangling combinations requiring repairing or substitutions.

For this purpose, we studied the encodings proposed by two well-known EAs: classic Genetic Algorithm (GA) and Genetic Programming (GP). We discarded the use of the classic GA

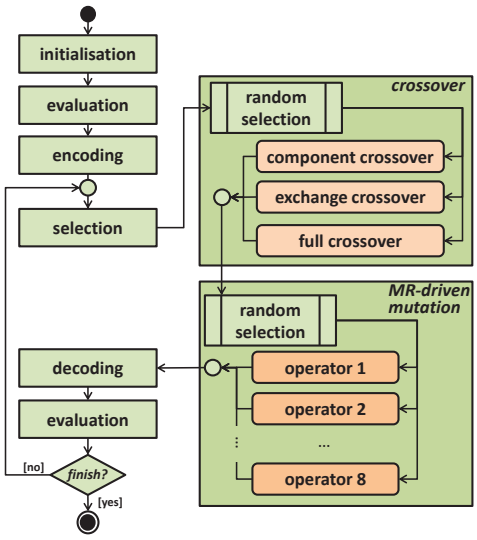


Figure 3: Main steps of the proposed evolutionary algorithm.

encoding because, even though it allows describing the cloud components, it has limitations to represent their interconnections. However, in GP, the relations between the different features of the models can be described using a tree encoding, which helps in representing the network connections. Nonetheless, we have identified several drawbacks complicating some steps of the algorithm. The first one is related to performance issues: the crossover and mutation steps require creating new versions of the cloud model, which is a computationally expensive task. Second, since we use MT to reduce the search space of the EA, it is necessary to use optimised structures for representing the individuals. Hence, we avoid the generation of dangling individuals to improve the overall performance of the algorithm.

For these reasons, we have designed a hybrid encoding based on graphs and integer representations, which facilitates the management of the large structures conforming the cloud, and reducing the generation of incorrect models. This way, we encode a cloud using two different vectors. The first one represents the physical components of the cloud (the racks), and the second one the network connections. For this purpose, we use the tuple-based representation for clouds explained in Section 5.1.

Example. Figure 4(a) shows the encoding of a cloud made of 4 racks (R0..R3), each one allocating a blade node, and 5 network connections (C0..C4). Rack R0 has a single board with one node. This node has a CPU with a speed of 15000 MFlops and 4 cores, a disk with 16000 GBytes providing a write and read bandwidth of 970 and 850 MBps, respectively, and a RAM memory of 8 GBytes providing a write and read bandwidth of 6000 MBps. C0 represents a network connection between racks R0 and R1, providing a bandwidth of 5000 MBps and latency of 5 μs. The genotype, i.e., the vectors containing the racks and the connections of

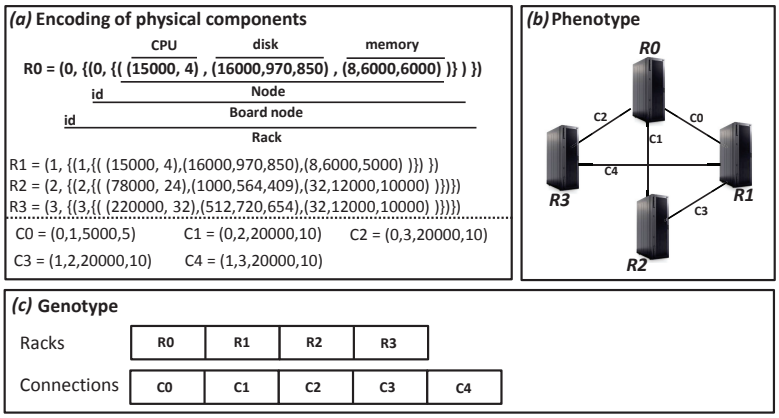


Figure 4: Encoding a cloud system: genotype and phenotype.

the cloud, is depicted in Figure 4(c). The phenotype, representing the topology of the cloud, is shown in Figure 4(b).

Selection. In this phase, the best individuals – based on an energy-aware criterion – have a higher probability to be selected for generating a new offspring. We use *roulette wheel* as a selection method, which gives high probabilities to those clouds models with low energy consumption. These probabilities are based on the width of the slice of a hypothetical roulette wheel, where wider slices provide higher probabilities to be selected.

Crossover. In this phase, individuals are combined to generate a new offspring following the principles of biological reproduction. Crossover mainly depends on the encoding and, therefore, using an appropriate crossover design improves the performance of the EA. Since the inclusion of multiple crossover mechanisms prevents the premature convergence and improves the performance of the algorithm [72], we propose three crossover techniques focused on specific knowledge of cloud systems, where for each iteration, the technique to be applied is randomly selected. The crossover operators we use are explained next:

- *Mix crossover*: This crossover merges the information of two physical machines to generate a new offspring. Initially, the tester must provide the percentage of machines involved in the crossover. Hence, one part of the cloud will remain unmodified, unless the tester indicates a 100%. Next, for each pair of clouds (parents) from the actual population, two new individuals (offspring) are generated. For this purpose the components of the physical machine are encoded in binary. Hence, the information of each component (CPU, disk, memory, ...) from both parents is combined using the standard one-point crossover. Figure 5 illustrates this crossover, where $CPU1$ (blue) and $CPU2$ (red) are combined to generate two new CPUs, each one containing information from both parents.
- *Swap crossover*: In this case the operator combines unmodified components from the parents to generate the offspring. Hence, the binary information of each component is not modified but placed in another individual. Similar to the mix crossover, the tester must

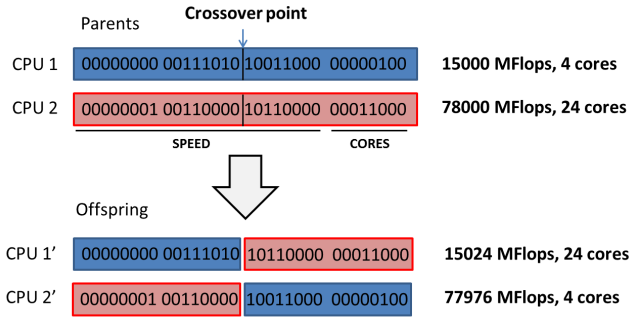


Figure 5: Example of mix crossover.

specify the percentage of physical machines involved in the crossover. Thus, the machines of the new offspring are a combination of components from both parents.

- *Full crossover*: This crossover is inspired by a NASA contribution to evolve graph topologies [73]. The idea is to randomly divide two clouds, each one in two fragments, and combine them to generate two new individuals. Figure 6 shows an example, where two different graphs $G1$ and $G2$ are combined to generate two new individuals, $G1'$ and $G2'$. To allow recombination, the graphs $G1$ and $G2$ are partitioned in such a way that $G1_a$ and $G2_a$ have the same number of edges cut (2 in this case).

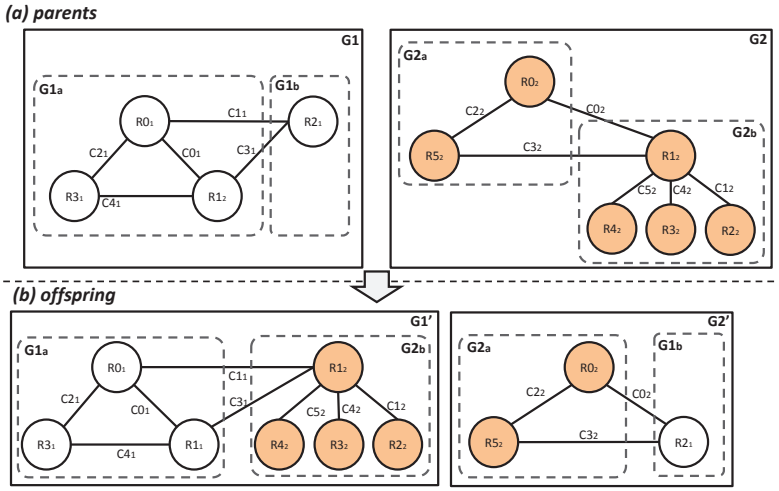


Figure 6: Example of full crossover.

MR-driven mutation. In this step, random mutations are seeded into the individuals. The idea is to explore those cloud models that provide better energy consumption without enhancing its

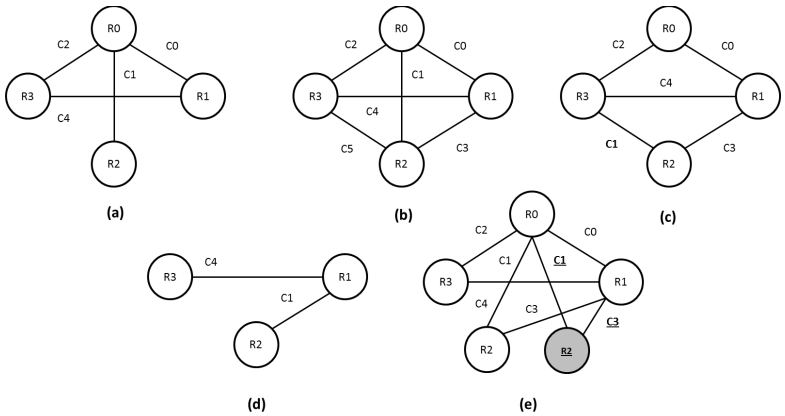


Figure 7: Example of mutation operator faults: (a) Operator 4 deletes link C3. (b) Operator 5 created link C5. (c) Operator 6 replaces the source of $C1$ by $R3$. (d) Operator 7 deletes $R0$. (e) Operator 8 splits $R2$.

underlying components. In some cases, a mutation is applied towards degenerating a specific part of the individual. Thus, the new individual represents a cloud model containing a-priori "worse" components than the source individual (e.g. slower CPUs, smaller memories or a data-centre containing less physical machines). The changes are performed with a certain probability, where small modifications have greater chances to be performed than larger ones, and are seeded in a way that the new generated individuals satisfy the MRs. This is one of the salient features of this work, where the MRs conduct the search towards a reduced exploration space. We have designed the following set of 8 mutation operators:

- *Operator 1*: This operator mutates all the components of a rack. Initially, a rack is randomly selected from a given cloud (individual), where all the racks have the same probability to be selected. Then, the operator seeds changes on all the components of the rack (nodes, disk, CPUs, ...). The following example illustrates how this operator is applied to the cloud represented in Figure 4. $R2$ is the rack before the operator is applied, and $R2'$ after its application. The changes are remarked in boldface. In this case, the mutation is applied towards degenerating the source individual.
  - $R2 = (2, \{(2, \{((78000, 24), (1000, 564, 409), (32, 12000, 10000))\})\})$
  - $R2' = (2, \{(2, \{((\mathbf{57200}, \mathbf{13}), (1000, \mathbf{364}, \mathbf{209}), (32, \mathbf{4212}, \mathbf{3210}))\})\})$
- *Operator 2*: This operator decreases the bandwidth of a network link of the cloud that is randomly selected from the communication network.
- *Operator 3*: Similarly to the previous operator, this one increases the latency of a randomly selected network link.
- *Operator 4*: This operator removes a link of the network, and is only applied if the resulting graph is connected. Figure 7 (a) illustrates the result of applying this mutation operator to the model of Figure 4, where link $C3$ is removed.
- *Operator 5*: This operator creates a new link connecting two (randomly selected) racks that are not directly connected. The bandwidth and latency values of the new network link

are set to the average of the existing links in the network. Figure 7 (b) shows the result of applying this mutation operator to the initial model, where the link C5 is added.

- *Operator 6*: This operator replaces the source/destination of a randomly selected link. An example application of this operator is depicted in Figure 7 (c), where the source of the link $C1$ ($R0$) is replaced by $R3$.
- *Operator 7*: This operator removes a randomly selected rack and all its connected links. If the resulting graph is unconnected, new links are created until the graph becomes connected. New connections are created based on the bandwidth and latency values of the removed connections. Destination nodes are randomly selected following the guidance of Operator 6. For this, it modifies the connection links to avoid unconnected graphs. Figure 7 (d) shows an example application, where R0 and all its connections are removed.
- *Operator 8*: This operator splits a rack in two, both having half of the capacities of the original one, and where the links are duplicated. An example application is shown in Figure 7 (e), where rack R2 and its links have been duplicated.

Decoding. A decoding process is required to evaluate the fitness of new and modified individuals. Although the crossover and mutation operators have been designed to avoid incorrect models, it is possible that some of the individuals do not satisfy some of the MRs provided as input, due to the multiple changes seeded in each model. In this case, the individual is replaced by another one that satisfies the MRs.

Please note that, as Figure 3 shows, after decoding, the individuals are evaluated again and the next population generated using those individuals with the lowest energy consumption.

7. Tool support

We have developed tool support for `MT-EA4Cloud`, which implements the different modules depicted in Figure 1 using Java. Its scheme is depicted in Figure 8, and shows how MT, EAs and simulation tools are combined to optimise the energy consumption in cloud systems. For the sake of clarity, only the most relevant parts are shown in this scheme.

The Evaluation Module (EM) consists of 4 main submodules: test case extraction module (labeled as A), template transformation module (labeled as B), cloud simulation module (labeled as C) and energy consumption module (labeled as D).

Initially, the *cloud chromosome* (in short, CCM) is extracted from the population. A CCM consists of three main elements: a *cloud model* (in short, CM), a *test case* (in short, TC) and *energy consumption* (in short, EC). Test cases are automatically generated using metamorphic testing techniques. Each TC contains a CM and a workload, where CM represents all the components used to model a cloud system, such as computational resources, network topology, and the workload refers to the operations to be processed by CM. Each object that represents a TC contains a path indicating the location of the test input data and a path where the results of the simulation are stored. The last element of the CMM, labeled as *Energy consumption*, refers to the amount of energy required by CM to execute the workload.

The information allocated in TC is parsed from the CCM to create a generic data structure. It is *generic* since it allocates the data required to configure a test case, but is not specific to a single simulator. The idea is to manage a common structure that can be applied to the different

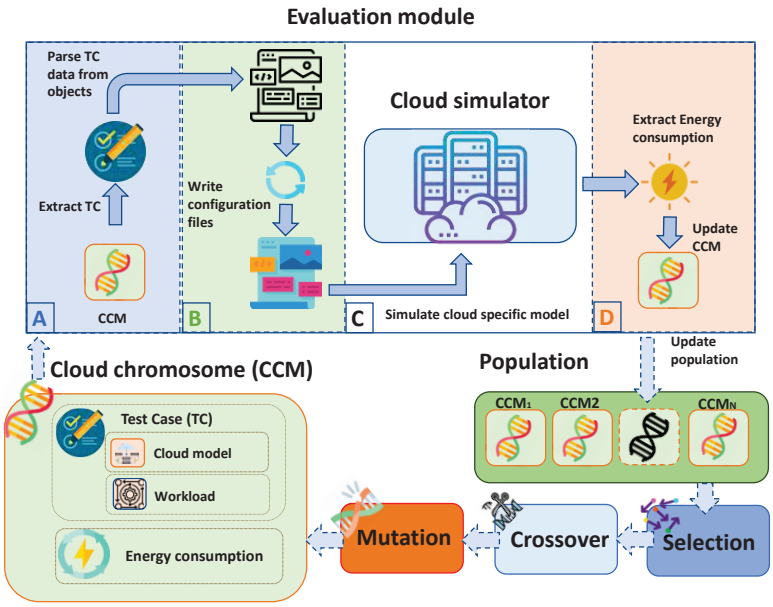


Figure 8: Tool support for `MT-EA4Cloud`: integration of the simulation tools within the EA scheme.

simulators deployed in the system, allowing an easy translation of this data into the different formats of the cloud simulators that are currently deployed (module A).

Next, the configuration files required to execute TC using a cloud simulator are generated (module B). The cloud simulator deployed in module C executes the simulation of the CM contained in TC. Once the simulation has finished, a results file is generated (see module D). The energy consumption is extracted from the results, and the CCM is updated and inserted into the population accordingly. The optimisation of the clouds is performed using the EA explained in Section 6, and depicted in Figure 3.

This framework has been developed using a modular and flexible design, which allows to easily include new simulators and other approaches inspired by EAs to optimise the energy consumption in clouds. For this, the main submodules of the EM have been represented as Java interfaces in such a way that the integration has been reduced to the following three points:

- Implementing the interface *TestCaseTransformations* (see module B), to transform a cloud model to the specific format required by the new simulator.
- Implementing the interface *SimExecution* (see module C), to provide the specific format to execute the simulations in the new integrated simulator.
- Implementing the interface *EnergyExtraction* (see module C), to parse the results file generated by the execution of the new simulator.

8. Empirical study

This section reports on an empirical study, where `MT-EA4Cloud` is applied to check the correctness of different cloud systems. First, in Section 8.1, we formulate several research questions we aim at answering with the experiments. Second, we present a detailed description of the experimental setting in Section 8.2. Next, in Section 8.3, we evaluate the adequacy of each MR in our catalogue and the suitability of each simulator used in this study. In Section 8.4, we perform a testing process to evaluate the correctness of different clouds using our approach. Finally, we discuss the obtained results and answer the research questions in Section 8.5.

8.1. Research questions

The experiments described in this section seek to answer the following questions:

- **RQ1**: *Is it feasible to analyse the correctness of energy-aware cloud systems using simulation?*

  In general, the main difficulty of choosing a tool that properly represents the underlying behaviour of the cloud lies in the uncertainty of the provided results, that is, how the researcher can be sure that the provided results properly represent the expected behaviour of the cloud? Hence, we are interested in analysing and comparing the suitability of different simulation tools to represent the features established by the user and, thus, to decide which simulation tool is most adequate to model and simulate these features.

- **RQ2**: *How adequate are the MRs for analysing energy-aware clouds?*

  In MT, the MRs model de underlying behaviour of the system under test. In this work, we provide a catalogue of MRs exclusively dealing with the energy consumption of cloud systems. Hence, we are interested in investigating the adequacy – from an energy-aware point of view – of the proposed MRs for studying cloud systems.

- **RQ3**: *Is it possible to automatically detect drawbacks in the energy consumption of cloud systems and provide convenient solutions?*

  We are interested in evaluating whether our proposed methodology is capable of optimising the energy consumption of the clouds under test by locating drawbacks in their underlying architectures. Thus, we say that a drawback is discovered in a cloud model when `MT-EA4Cloud` locates an alternative cloud model that provides better energy consumption without enhancing its underlying components, like increasing the CPU speed, using more physical machines in the data-centre or using a faster network. Additionally, we are interested in comparing the quality of the test cases generated using our proposed EA against the quality of randomly generated test cases.

8.2. Experimental setting

The main objective of our methodology is to check the correctness – from an energy consumption point of view – of cloud systems. Hence, the first step consists in analysing the cloud features having a direct impact on energy consumption, which are modelled in the form of MRs. In this study, the catalogue of MRs presented in Section 5.2 has been used to model the underlying behaviour of cloud systems formally.

In order to execute the experiments of this study, we have chosen different well-known simulators designed to model and simulate the energy consumption of cloud systems. First, we have

carefully investigated, in the research papers found in the current literature, the main features and drawbacks of each simulator. Second, we have analysed the documentation provided by each simulator to decide whether a simulator is appropriate, or not, for this study. As a result, we provide the set $\mathcal{S}$ consisting of 7 simulators (step 2). Table 1 shows the MRs of the catalogue that can be modelled and simulated using each simulator $S \in \mathcal{S}$.

Id	MR_1	MR_2	MR_3	MR_4	MR_5	MR_6	MR_7	MR_8
CloudSim	✓	✓	✓	✗	✓	✗	✓	✓
CloudSimStorage	✓	✓	✓	✓	✓	✗	✓	✓
DCSIM	✓	✓	✓	✓	✓	✗	✓	✓
GreenCloud	✓	✓	✓	✗	✓	✗	✓	✓
SimGrid	✓	✓	✓	✗	✓	✗	✓	✓
iCanCloud	✓	✓	✓	✓	✓	✓	✓	✓
DISSECT-CF	✓	✓	✓	✓	✓	✗	✓	✓

Table 1: Analysis of different cloud simulators to represent the provided MRs.

Next, we manually design three clouds – *cloudA*, *cloudB* and *cloudC* – providing different configurations, each one representing a specific cloud profile. Thus, *cloudA* represents a low-profile cloud, providing slow CPUs, small RAM memories and a slow network; *cloudB* models a high-profile cloud, with large RAM memories, fast CPUs with 8 cores and a fast communication network; and *cloudC* represents a mid-profile cloud, with a fast communication network and a fair CPU and memory system. The configurations of these clouds are depicted in Table 2.

Additionally, we have created different workloads, which are inspired by operations performed in big data analysis. In particular, we use traces that represent the infrastructure of PlanetLab [74], to be executed by cloudSim, and a Map-Reduce based application [75], to be executed by simGrid. It must be noted that each simulator requires a specific type of application to be executed and, therefore, the same application cannot be executed in both simulators. Hence, the idea is to apply different workloads over different cloud systems to analyse the energy consumption in these systems. In the following, a trace is denoted by ω_{sim}^{size}, where *sim* is the simulator used to execute the trace and *size* represents the trace length. The size of a small trace is denoted by the sub-index *s*, the size of a medium trace – larger than the small trace – is denoted by *m*, and the size of the largest trace is denoted by *l*.

The set of source test cases, denoted by $\mathcal{T}$, is generated by combining the clouds presented in Table 2 and the three generated workloads. In step 4, we automatically generate a set of follow-up test cases using C and $\mathcal{T}$, as input. In this case, we generate the set $\mathcal{F}$ containing a total of

Parameter / Cloud	*cloudA*	*cloudB*	*cloudC*
#Hosts	512	512	512
RAM (MBytes)	1024	16384	8192
CPU speed (MIPS)	1k	90k	20k
CPU cores	2	8	4
HDD size (TBytes)	1	1	1
HDD speed (Mbps)	20	350	100
Net bw (Mbps)	500	10000	10000
Net lat (us)	10	10	10

Table 2: Source cloud configurations.

4000 follow-up test cases.

8.3. Assessing the effectiveness of the MRs and the suitability of the simulators

In this section, the catalogue of MRs C (see Section 5) is evaluated using equation 1 to calculate the adequacy. This requires an MR, a simulator $S \in \mathcal{S}$ and a set of source test cases $\mathcal{T}$. We have calculated the adequacy for each pair $(MR, S) \in C \times \mathcal{S}$. The results are presented in Table 3, where each column refers to an MR, and each row represents a simulator. In essence, these results show the percentage of tests that fulfil the different MRs for each simulator.

Id	MR_1	MR_2	MR_3	MR_4	MR_5	MR_6	MR_7	MR_8
$CloudSimStorage$	100	40	100	100	99	-	100	100
$GreenCloud$	0	20	100	-	45	-	100	100
$SimGrid$	100	63	100	-	100	-	100	100
$iCanCloud$	100	39	100	100	89	84	73	100
$DISSECT-CF$	99	51	100	99	95	-	100	100

Table 3: Adequacy (in %) of each MR using different simulators.

In general, all simulators used in this study provide acceptable results to simulate cloud systems. However, there are some simulators that cannot model the features formulated in the MRs. For instance, GreenCloud does not provide capabilities to model MR_4 and MR_6. In other cases, the obtained results show that some simulators do not properly represent the expected behaviour of a cloud. For example, in the case of iCanCloud, it achieves low effectiveness in MR_2 and MR_7, while the rest of the simulators provide better results for these MRs. It is important to remark that the catalogue of MRs is designed to represent a general view of the cloud and, thus, there are some specific situations where the MRs cannot reflect the real behaviour of the cloud. These situations are represented when some follow-up test cases do not satisfy a given MR. For instance, when cloudSim-storage is used to simulate the follow-up test cases generated to evaluate MR_5, which obtains a 99% of adequacy score. In this case, only 1% of the cases cannot be represented by this MR.

As the table shows, MR_2 provides a low adequacy score, that is, a high number of test cases do not fulfil this MR. This means that none of the simulators used in this study provides significant results supporting the claim that MR_2 accurately represents the behaviour of a cloud system. Consequently, since this MR cannot be used in the testing process, it has been discarded.

Table 3 also shows that MR_6 can be only modelled using the iCanCloud simulator. Usually, cloud simulators do not provide proper models for calculating the energy consumption in the memory system. Although iCanCloud implements this feature, the obtained results are not as promising as the ones obtained for evaluating other systems, like the computing system or the storage system.

Another feature that must be taken into account is the performance provided by the simulator. Since the methodology presented in this paper requires to execute a large number of simulations, this aspect must be carefully taken into account. For example, the iCanCloud simulator requires almost 1 hour to simulate a small scenario, while simGrid provides the results in a few minutes.

Once all the simulators have been checked (step 5), we analyse the results presented in Table 3 to select the most appropriate simulators for the testing process. If we discard MR_2 and MR_6, we notice that cloudSimStorage and simGrid provide results achieving, at least, 99% of adequacy in the rest of MRs. Hence, we chose cloudSimStorage and simGrid as the most appropriate

simulators, for two reasons. First, they provide high performance for executing the simulations. Second, the obtained results show that these simulators are suitable to model and simulate the required features of cloud models, represented in each $MR \in C$.

8.4. Checking the correctness of energy-aware cloud systems

In this section, we perform a set of experiments to analyse the correctness of the clouds designed in step 3. In essence, we test a cloud by measuring the required amount of energy to execute a workload. The testing process is conducted by using two different techniques, our proposed combination of MT and EAs, and a random approach. Both techniques focus on automatically generating test cases, where a test case consists of a workload and a cloud configuration. Their main difference lies in how the cloud configuration is generated. While our approach performs a guided search that uses the "best" clouds of each generation to create a new offspring of individuals, the second approach generates new clouds by apply mutations randomly over the original cloud. It is important to remark that both approaches generate clouds that satisfy the input constraints of the MRs. Hence, since the former approach performs a guided search, we expect to outperform the random approach both in performance and efficiency.

Our EA evolves different generations of individuals using three different crossovers and eight mutation operators. The crossover to be applied to each individual is randomly selected, while we have tried three different probability levels (three configurations) for applying the mutation operators. As a remark, since each simulator requires a specific configuration to model the cloud, there are some mutation operators that cannot be applied in certain simulators. In order to alleviate this issue, we provide a specific configuration for each simulator.

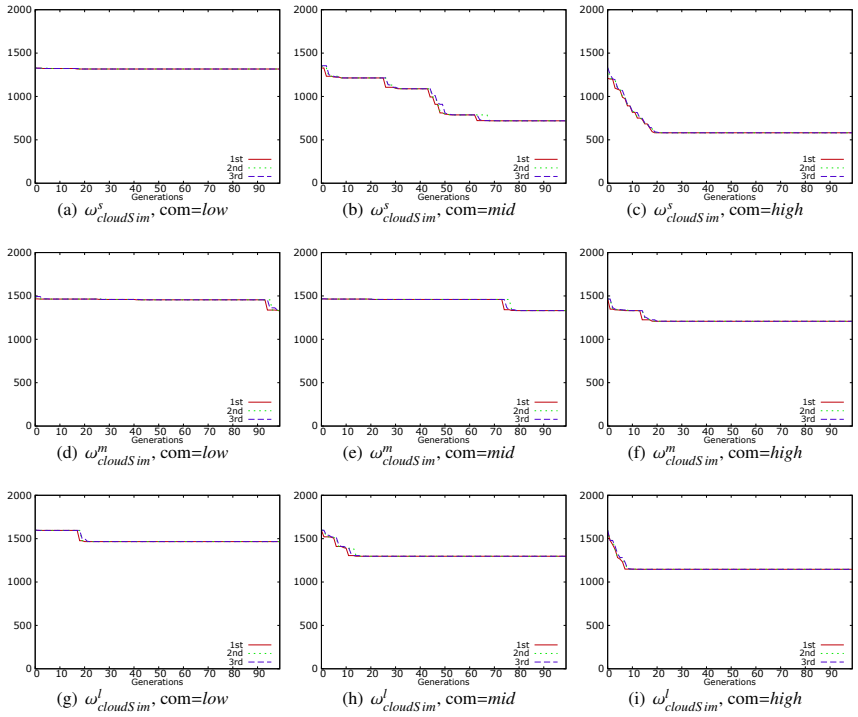
Table 4 shows the configurations – namely *low*, *mid* and *high* – used in the testing process when cloudSim executes the simulations. Each column of the table represents the probability of each mutation operator to be applied, where the sum of the probabilities must be less or equal than 100. In the cases where the sum of the probabilities is less than 100, we use the difference to represent the null operator, that is, none of the mutation operators is applied. On the contrary, when the sum of the probabilities is 100, one of the operators is mandatorily applied. For instance, using the *mid* configuration of Table 4, there is a probability of 100-(15+10+5+1)=69% to apply the null operator.

Config	$mutOp_1$	$mutOp_2$	$mutOp_3$	$mutOp_4$	$mutOp_5$	$mutOp_6$	$mutOp_7$	$mutOp_8$
low	1.5%	1.5%	1%	–	–	–	0.5%	–
mid	15%	10%	5%	–	–	–	1%	–
high	25%	25%	25%	–	–	–	25%	–

Table 4: Configuration to apply the mutation operators in cloudSim.

Figure 9, 10 and 11 show the results obtained in the testing process using cloudSim to execute the simulations and the individuals *cloudA*, *cloudB* and *cloudC* as a seed to generate the initial population, respectively. Each figure contains 9 charts, corresponding to the combination of executing three different workloads using three different configurations. Thus, the charts of the same row represent the simulations that use the same workload, while the charts of the same column represent the simulations that use the same configuration for applying the mutation operators (in short, *com*). The x-axis of each chart represents the generations of individuals (clouds) and the y-axis represents the energy consumption of each cloud, measured in kW. Each chart shows the three best individuals from each generation. The idea is to investigate how the

best individuals of each generation evolve and, thus, to analyse how their genetic information – transferred to further generations – affects the energy consumption. The best individual of each generation is shown in red, the second one in green and the third one in blue.



(a) $\omega^{s}_{cloudSim}$, com=*low* (b) $\omega^{s}_{cloudSim}$, com=*mid* (c) $\omega^{s}_{cloudSim}$, com=*high*

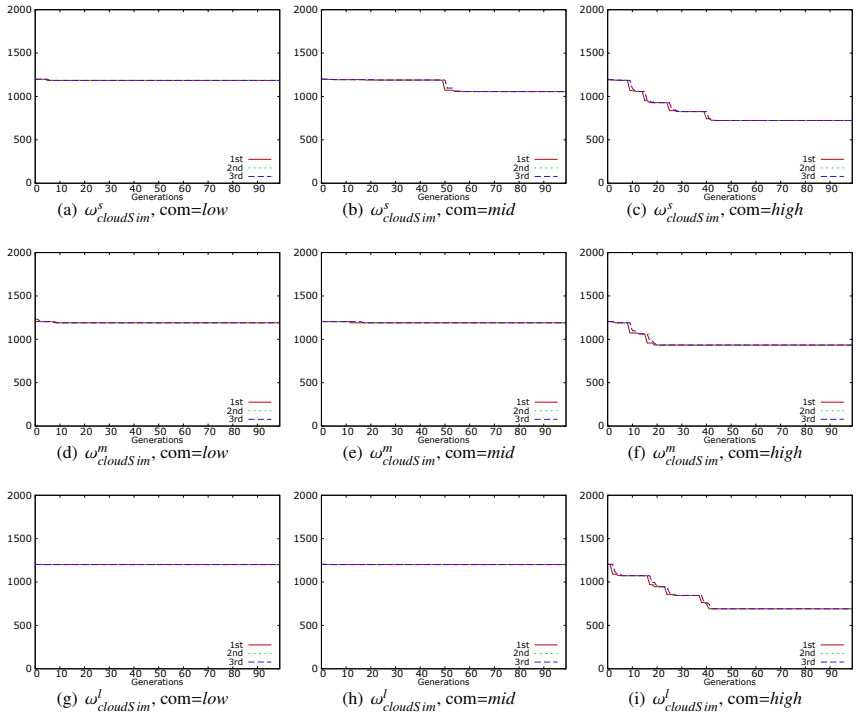
(d) $\omega^{m}_{cloudSim}$, com=*low* (e) $\omega^{m}_{cloudSim}$, com=*mid* (f) $\omega^{m}_{cloudSim}$, com=*high*

(g) $\omega^{l}_{cloudSim}$, com=*low* (h) $\omega^{l}_{cloudSim}$, com=*mid* (i) $\omega^{l}_{cloudSim}$, com=*high*

Figure 9: Testing process using cloudSim and *cloudA* as a seed to generate the initial population.

Figure 9 shows that the configuration used for mutating the individuals has a significant impact on the quality of the further generations. That is, using a configuration with a low mutation probability provides a slight improvement in the offsprings, while using a configuration with a high mutation probability provides substantial improvement. This fact is appreciated in the charts of the same row, which use the same workload. We can also notice that the workload also has an impact on energy consumption. In this case, the larger workload $\omega^{l}_{cloudSim}$ requires more energy to be processed than the other workloads. These results also show that the tendency of each generation, in the sense of improving the energy consumed by the cloud, is practically the same. This can be appreciated in how the best three individuals of each generation are similarly improved along with the generations.

Figure 10 shows that the *low* configuration provides similar results for all the workloads. The *mid* configuration enhances *cloudB* only for the smallest workload, that is, $\omega^{s}_{cloudSim}$ (see



(a) $\omega^{s}_{cloudSim}$, com=*low* (b) $\omega^{s}_{cloudSim}$, com=*mid* (c) $\omega^{s}_{cloudSim}$, com=*high*

(d) $\omega^{m}_{cloudSim}$, com=*low* (e) $\omega^{m}_{cloudSim}$, com=*mid* (f) $\omega^{m}_{cloudSim}$, com=*high*

(g) $\omega^{l}_{cloudSim}$, com=*low* (h) $\omega^{l}_{cloudSim}$, com=*mid* (i) $\omega^{l}_{cloudSim}$, com=*high*

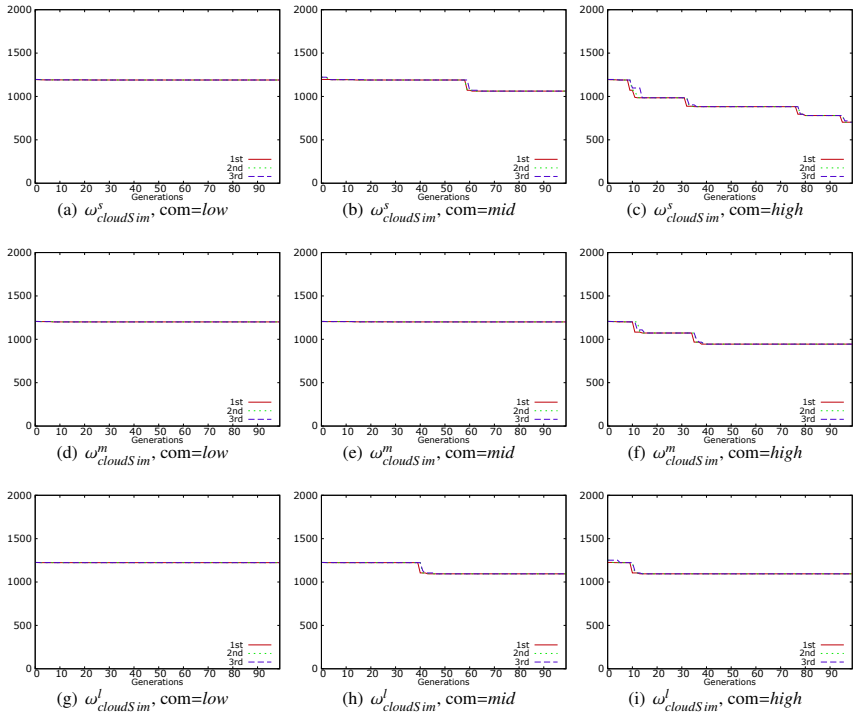
Figure 10: Testing process using cloudSim and *cloudB* as a seed to generate the initial population.

Figure 10.b). On the contrary, the *high* configuration provides notable improvements along the generations. In this case, although using the *high* configuration provides promising results, there is a corner case – using workload $\omega^{m}_{cloudSim}$ – where the EA is not able to find a proper optimization of the cloud (see Figure 10.f). This effect can be observed by looking at Figure 10.i, which shows that the EA provides an enhanced cloud for executing $\omega^{l}_{cloudSim}$, which theoretically must require more energy than the execution of $\omega^{m}_{cloudSim}$.

Similarly, Figure 11 depicts that the *high* configuration is more suitable to enhance the initial cloud than the rest of the configurations (see Table 4). In this case, when using the *mid* configuration, the EA slightly enhances the initial cloud only for processing $\omega^{s}_{cloudSim}$ and $\omega^{l}_{cloudSim}$. On the contrary, when the *high* configuration is used, the obtained improvement is inverse in proportion to the size of the processed workload (see Figure 11.c, 11.f and 11.i).

Figures 12, 13 and 14 show the results of applying our EA to *cloudA*, *cloudB* and *cloudC* using the simGrid simulator. Table 5 shows the configurations used in this experiment.

When the EA is applied to *cloudA* (see Figure 12), we observe that the energy consumption of the cloud is proportional to the size of the executed workload. Although this fact can also



(a) $\omega^{s}_{cloudSim}$, com=*low* (b) $\omega^{s}_{cloudSim}$, com=*mid* (c) $\omega^{s}_{cloudSim}$, com=*high*

(d) $\omega^{m}_{cloudSim}$, com=*low* (e) $\omega^{m}_{cloudSim}$, com=*mid* (f) $\omega^{m}_{cloudSim}$, com=*high*

(g) $\omega^{l}_{cloudSim}$, com=*low* (h) $\omega^{l}_{cloudSim}$, com=*mid* (i) $\omega^{l}_{cloudSim}$, com=*high*

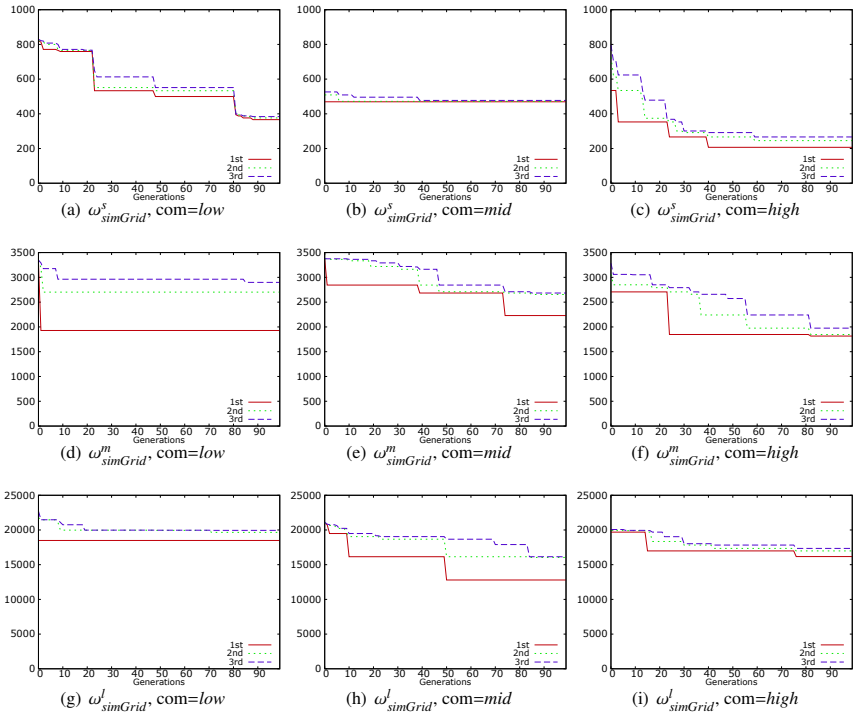
Figure 11: Testing process using cloudSim and *cloudC* as a seed to generate the initial population.

Config	$mutOp_1$	$mutOp_2$	$mutOp_3$	$mutOp_4$	$mutOp_5$	$mutOp_6$	$mutOp_7$	$mutOp_8$
low	1%	1%	1%	0.25%	0.25%	0.25%	1%	0.25%
mid	10%	10%	5%	1%	1%	1%	5%	1%
high	20%	20%	20%	5%	5%	5%	20%	5%

Table 5: Configuration to apply the mutation operators in simGrid.

be appreciated when the simulations are executed using the cloudSim simulator, in this case, the difference between the energy consumptions is more significant. For instance, executing $\omega^{m}_{simGrid}$ requires 5 times more energy than the execution of $\omega^{s}_{simGrid}$ (see Figure 12.a and 12.d). Moreover, increasing the probability to perform a mutation in the individuals provides a greater improvement in the offsprings than using a low probability. Nevertheless, there is an exception in Figure 12.d, where using the *low* configuration provides almost the same results than using a higher probability to perform the mutations (see Figure 12.f). This is caused due to the stochastic nature of the algorithm.

Figure 13 depicts that the EA provides a similar tendency in the energetic consumption, using



(a) $\omega^{s}_{simGrid}$, com=*low* (b) $\omega^{s}_{simGrid}$, com=*mid* (c) $\omega^{s}_{simGrid}$, com=*high*

(d) $\omega^{m}_{simGrid}$, com=*low* (e) $\omega^{m}_{simGrid}$, com=*mid* (f) $\omega^{m}_{simGrid}$, com=*high*

(g) $\omega^{l}_{simGrid}$, com=*low* (h) $\omega^{l}_{simGrid}$, com=*mid* (i) $\omega^{l}_{simGrid}$, com=*high*

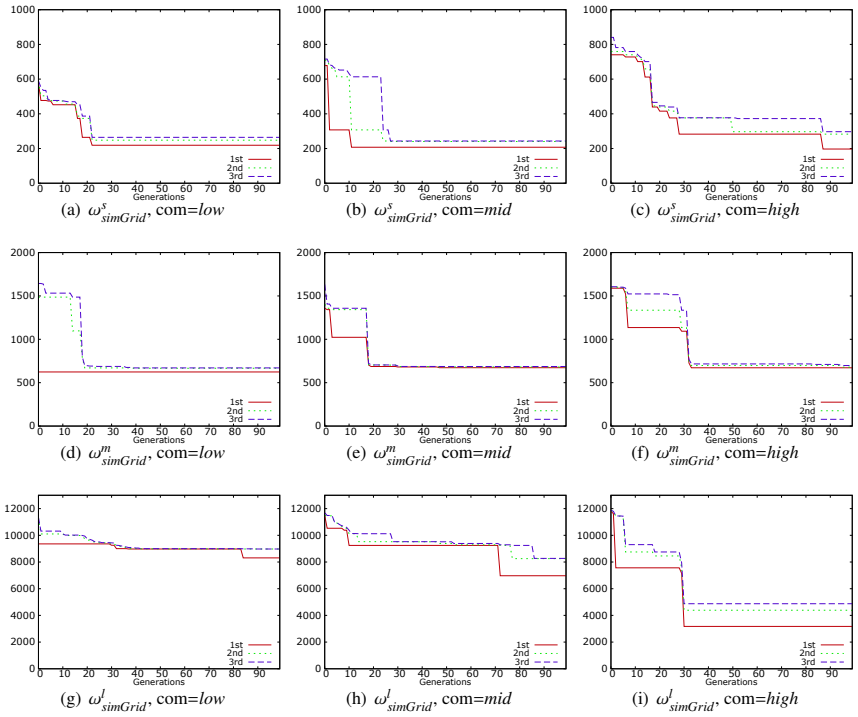
Figure 12: Testing process using simGrid and *cloudA* as a seed to generate the initial population.

all the configurations for applying mutation operators, when $\omega^{s}_{simGrid}$ and $\omega^{m}_{simGrid}$ are processed. However, when the cloud processes $\omega^{l}_{simGrid}$ using the *high* configuration, the obtained cloud configuration significantly improves the energy consumption (see Figure 13.i). It is important to remark that, in this experiment, the corner case shown in Figure 12 is not present.

The charts depicted in Figure 14 show similar results than the ones obtained in the previous experiments. However, in this case, we identify a significant difference in the energy consumption between the first three individuals of each generation (see Figure 14.i). On the contrary, the rest of the cases show a relatively similar improvement for the best three individuals.

Overall, the results obtained in these experiments show that the *high* configuration provides the best cloud optimizations. Also, this configuration is the fastest to find a proper optimization of the cloud, that is, the final result is reached by creating fewer generations of individuals than the other configurations. However, there are some corner cases where the EA is not able to find an appropriate optimization of the cloud. In any case, the reached solution outperforms, in the sense of energy consumption, the initial cloud configuration.

In order to check the effectiveness of our EA, we present an experiment comparing the best



(a) $\omega^{s}_{simGrid}$, com=*low* (b) $\omega^{s}_{simGrid}$, com=*mid* (c) $\omega^{s}_{simGrid}$, com=*high*

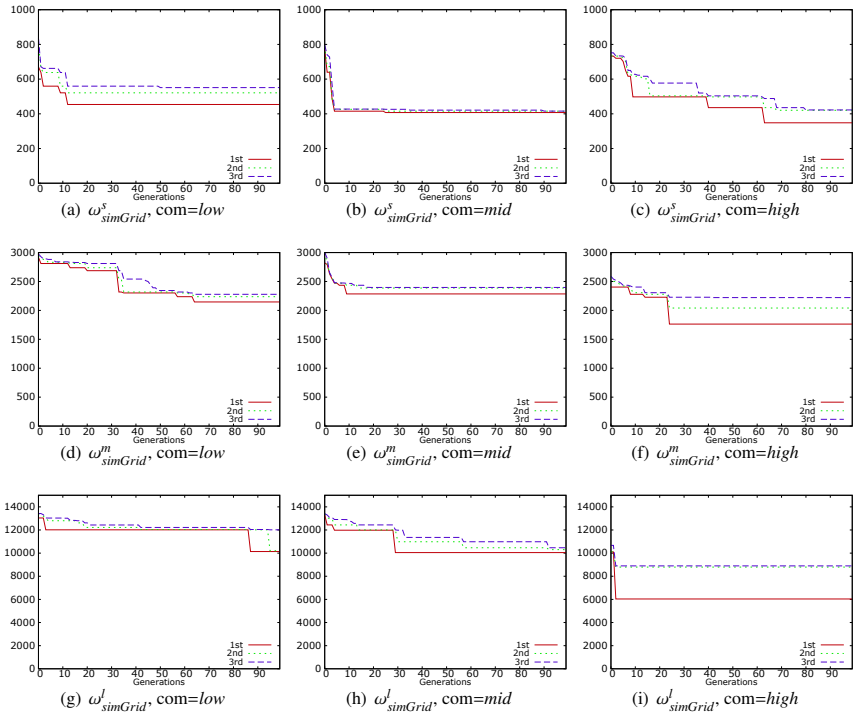
(d) $\omega^{m}_{simGrid}$, com=*low* (e) $\omega^{m}_{simGrid}$, com=*mid* (f) $\omega^{m}_{simGrid}$, com=*high*

(g) $\omega^{l}_{simGrid}$, com=*low* (h) $\omega^{l}_{simGrid}$, com=*mid* (i) $\omega^{l}_{simGrid}$, com=*high*

Figure 13: Testing process using simGrid and *cloudB* as a seed to generate the initial population.

cloud optimization obtained by the EA against the results obtained by an approach that randomly applies mutation operators to generate new cloud configurations. From now on, we refer to the second approach as *random approach*. In both approaches, the MRs are used to check the new generated clouds, which must fulfil the constraints reflected in the input part of each MR.

Figure 15 shows the results obtained for simulating, using cloudSim, the executions of the workloads $\omega^{s}_{simGrid}$, $\omega^{m}_{simGrid}$ and $\omega^{l}_{simGrid}$ over *cloudA*, *cloudB* and *cloudC*. In this experiment, for each cloud, 150 different configurations are generated. Each new configuration is generated by applying one mutation operator, which is randomly selected, over the initial cloud. The x-axis of each chart represents the generated cloud configurations (individuals), and the y-axis represents the energy consumption of each cloud. The red line shows the energy consumption of the new generated clouds, while the green line represents the best cloud configuration reached by the EA. In this figure, each row of charts represents the same cloud configuration, while each column of charts represents the same workload. The EA clearly provides the best result in all cases. The random approach provides similar results for each cloud using all the workloads. However, the charts show some noticeable peaks, which become more significant when the size of the pro-



(a) $\omega^{s}_{simGrid}$, com=*low* (b) $\omega^{s}_{simGrid}$, com=*mid* (c) $\omega^{s}_{simGrid}$, com=*high*

(d) $\omega^{m}_{simGrid}$, com=*low* (e) $\omega^{m}_{simGrid}$, com=*mid* (f) $\omega^{m}_{simGrid}$, com=*high*

(g) $\omega^{l}_{simGrid}$, com=*low* (h) $\omega^{l}_{simGrid}$, com=*mid* (i) $\omega^{l}_{simGrid}$, com=*high*

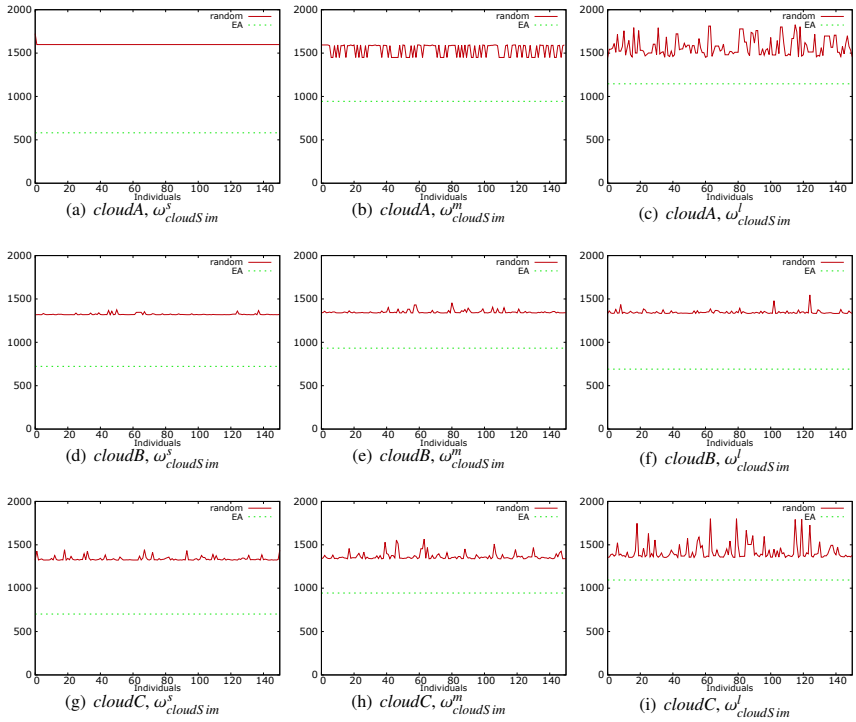
Figure 14: Testing process using simGrid and *cloudC* as a seed to generate the initial population.

cessed workload increases. In general, the difference between the EA and the random approach is greater when the small workload is processed. On the contrary, the difference between these approaches is smaller when the large workload is executed in the clouds. There is one corner case (see Figure 15.e and Figure 15.f) where the EA provides a better result for processing a larger workload. However, the random approach provides almost the same result for these cases.

Figure 16 shows the results when the simGrid simulator is used to execute the workloads over the clouds. In this case, both approaches provide similar results for processing the small and medium workloads. Although in most of the cases the EA is better than the random approach, there is one case where the random approach outperforms the EA (see individual 120 in Figure 16.h). Similarly to the previous experiment, in this case, the difference between the results obtained by both approaches is more significant as the size of the processed workload increases.

8.5. Discussion of the results and answers to the research questions

In this subsection, we discuss the obtained results and answer the research questions presented in Section 8.1.



(a) *cloudA*, $\omega^{s}_{cloudSim}$ (b) *cloudA*, $\omega^{m}_{cloudSim}$ (c) *cloudA*, $\omega^{l}_{cloudSim}$

(d) *cloudB*, $\omega^{s}_{cloudSim}$ (e) *cloudB*, $\omega^{m}_{cloudSim}$ (f) *cloudB*, $\omega^{l}_{cloudSim}$

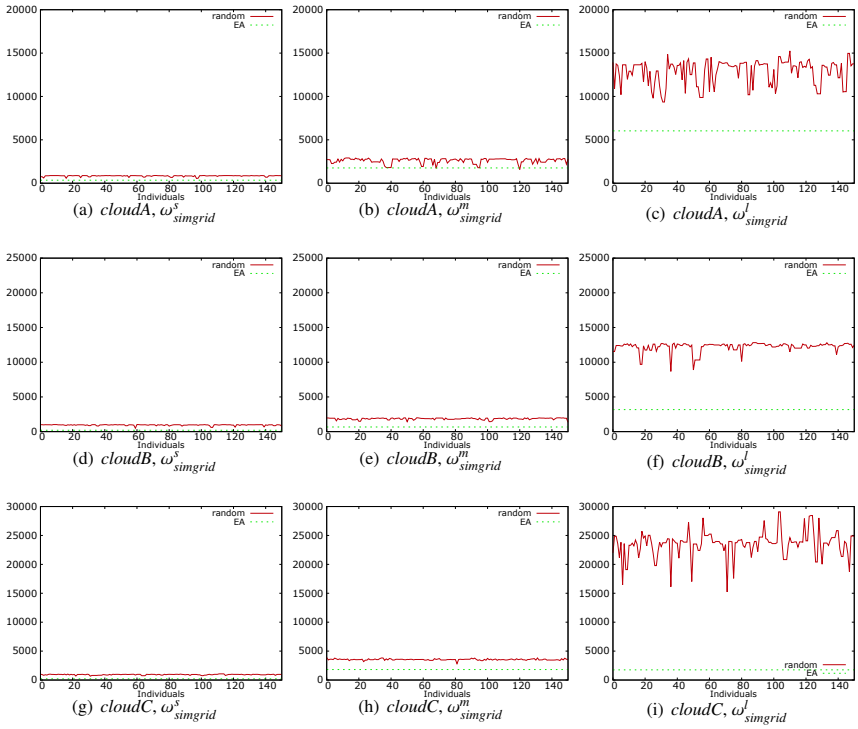
(g) *cloudC*, $\omega^{s}_{cloudSim}$ (h) *cloudC*, $\omega^{m}_{cloudSim}$ (i) *cloudC*, $\omega^{l}_{cloudSim}$

Figure 15: Testing process using cloudSim and randomly generated cloud configurations.

8.5.1. ***RQ1****: Is it feasible to analyse the correctness of energy-aware cloud systems using simulation?*

In order to answer this question, we have designed `MT-EA4Cloud`, a methodology to check the correctness – from an energy consumption point of view – of different cloud systems. We have carried out an empirical study, which has been supported by our proposed methodology, to check the suitability of seven simulators for modelling and simulating the energy consumption of three different cloud systems (see Section 8.2). The analysed features of the cloud have been modelled in the form of MRs (see Section 5.2). We identify those features that can be modelled for each simulator in Table 1 and, thus, we are able to decide if a given simulator appropriately simulates these features, which represent the underlying behaviour of the cloud. In this case, we have discarded two simulators from the initial list $\mathcal{S}$ and, therefore, the remaining simulators are used to carry out the testing process in the following experiments.

We can conclude that the answer to **RQ1** is *yes, it is feasible to analyse the correctness of energy-aware cloud systems using simulation*. However, there are some steps that must be manually performed by the expert, like designing an appropriate catalogue of MRs. Although



(a) *cloudA*, $\omega^s_{simgrid}$ (b) *cloudA*, $\omega^m_{simgrid}$ (c) *cloudA*, $\omega^l_{simgrid}$

(d) *cloudB*, $\omega^s_{simgrid}$ (e) *cloudB*, $\omega^m_{simgrid}$ (f) *cloudB*, $\omega^l_{simgrid}$

(g) *cloudC*, $\omega^s_{simgrid}$ (h) *cloudC*, $\omega^m_{simgrid}$ (i) *cloudC*, $\omega^l_{simgrid}$

Figure 16: Testing process using simGrid and randomly generated cloud configurations.

the major part of the work can be automated – generating a large number of follow-up test cases and simulating a wide spectrum of cloud models – it is key that the expert takes the right decisions in those steps requiring the user intervention.

8.5.2. ***RQ2****: How adequate are the MRs for analysing energy-aware clouds?*

In order to answer this question, we provide a complete catalogue of MRs that model the underlying behaviour of cloud systems regarding energy consumption (see Section 5.2). The adequacy of each MR is calculated using Equation 1 and shown in Table 3.

After a careful analysis of the results depicted in Table 3, we can conclude that *the catalogue of MRs is adequate to analyse energy-aware clouds*. First, the adequacy of each MR is calculated to discard those that do not properly model the underlying behaviour of a cloud environment. Second, the major part of the accepted MRs provides promising results reaching, at least, a 99% of adequacy score. Hence, designing accurate MRs is crucial for successfully applying our proposed methodology and, consequently, the quality of the results heavily depends on the accuracy of the MRs to model the underlying behaviour of the cloud.

8.5.3. ***RQ3****: Is it possible to automatically detect drawbacks in the energy consumption of cloud systems and provide convenient solutions?*

In order to answer this question, we have conducted an empirical study (see Section 8.4) where three different cloud systems have been analysed using two well-known simulators, simGrid and cloudSim. For the sake of clarity, we summarise the results obtained from this study in Tables 6 and 7. In essence, we use two different approaches to test the clouds. The first approach (labelled as EA) uses our EA to find a proper optimisation of the cloud under test, while the second approach (labelled as Random) randomly applies a mutation operator to the cloud under test for generating new cloud configurations.

Configuration		EA			Random		
Cloud	Workload	min	max	avg	min	max	avg
cloudA	$\omega^{s}_{cloudSim}$	702.99	1189.99	1001.20	1323.25	1443.94	1331.14
	$\omega^{m}_{cloudSim}$	944.52	1200.77	1114.78	1333.27	1566.17	1353.12
	$\omega^{l}_{cloudSim}$	1094.73	1223.34	1137.89	1354.37	1800.84	1412.05
cloudB	$\omega^{s}_{cloudSim}$	723.09	1184.74	987.65	1318.06	1374.83	1320.46
	$\omega^{m}_{cloudSim}$	933.82	1190.40	1104.28	1322.89	1455.99	1339.29
	$\omega^{l}_{cloudSim}$	691.72	1201.55	1030.47	1332.74	1546.55	1341.36
cloudC	$\omega^{s}_{cloudSim}$	581.43	1316.57	869.43	1449.42	1827.10	1498.49
	$\omega^{m}_{cloudSim}$	1207.75	1455.49	1318.36	1449.42	1598.15	1566.77
	$\omega^{l}_{cloudSim}$	1146.80	1466.18	1303.45	1395.83	2165.27	1817.95

Table 6: Pseudo-random versus EA approach using cloudSim.

Configuration		EA			Random		
Cloud	Workload	min	max	avg	min	max	avg
cloudA	$\omega^{s}_{simGrid}$	206.99	903.71	600.27	668.37	1065.15	939.68
	$\omega^{m}_{simGrid}$	1815.57	3500.75	3116.08	2825.67	3895.14	3551.99
	$\omega^{l}_{simGrid}$	1761.64	23863.45	16479.02	15223.84	29083.71	23419.71
cloudB	$\omega^{s}_{simGrid}$	196.95	806.33	540.37	489.79	1022.51	943.46
	$\omega^{m}_{simGrid}$	666.44	1882.89	988.29	1350.58	1997.83	1856.99
	$\omega^{l}_{simGrid}$	3174.85	11926.45	9479.64	8671.07	12823.77	12259.79
cloudC	$\omega^{s}_{simGrid}$	348.22	922.13	636.67	549.92	1009.09	845.64
	$\omega^{m}_{simGrid}$	**1763.21**	2724.51	2478.34	**1577.35**	2921.28	2617.96
	$\omega^{l}_{simGrid}$	6034.08	15257.92	12389.22	9118.56	15257.92	12829.67

Table 7: Pseudo-random versus EA approach using simGrid.

In the tables, the first column represents the cloud under test and the executed workload. The second column depicts the results provided by the EA, while the third column shows the results provided by the random approach. Each value represents the energy required by the cloud to execute the workload. Thus, *min* refers to the "best" cloud configuration generated, in the sense of energy consumption, using as a basis the cloud under test, *max* refers to the "worst" generated cloud and *avg* refers to the energy consumption average of all the generated cloud configurations – using the cloud under test as basis – to execute a specific workload.

It is important to remind that each simulator executes a specific type of workload (see Section 8.2) and, therefore, the results obtained from cloudSim cannot be compared with those obtained from simGrid.

First of all, we observe that our EA clearly provides better results than the random approach. There is only one exception, where the cloud configurations generated from *cloudC* execute $\omega^{m}_{simgrid}$ (see highlighted values in Table 7). In this particular case, the random approach generates a better optimization of the cloud because the EA is not able to find proper optimization during the first 100 generations.

These results also show that cloudSim provides similar results to execute the three workloads. However, the results provided by the simulations executed by simGrid show otherwise, where the energy consumption of the cloud is proportional to the size of the processed workload. We think that this is mainly caused by the accuracy of these simulators, that is, while cloudSim provides similar results to execute workloads of different sizes, simGrid clearly reflects a significant difference in the energy required to execute each workload.

Our proposed EA scales well when the size of the executed workload grows, as the optimization reached is more substantial when the largest workload is processed. Since the EA is able to dynamically adapt the size of the cloud – in number of physical machines – to execute a given workload, the generated cloud should be more optimised – in the sense of energy consumption – for executing this specific workload. On the contrary, the random approach only performs one modification to the cloud under test, which in some cases is not enough to reach a proper optimization of the cloud.

After a careful analysis of the results, we can conclude that the answer to **RQ3** is *yes, our methodology is able to provide different alternatives to improve the energy consumption and automatically detect flaws in the cloud designs created by the user*.

9. Threats to validity

In this section, we discuss the threats to validity of our empirical study.

9.1. Internal threats

Internal validity concerns whether our findings, which are based on the obtained results from the empirical study, truly represent a cause-and-effect relationship. Thus, the internal validity of our study lies in the implementation of our experiments.

The design of the provided catalogue of MRs is based on the experience of two experts. We are aware that the ability of MT to detect errors in the system highly depends on the selection of metamorphic properties and, therefore, the results may have varied if different MRs were used instead. Moreover, the use of domain-specific properties, like the ones used to design our proposed catalogue of MRs, should reveal a high failure percentage.

We have implemented both the EA and the MRs in Java. Also, we use different simulators, which have been widely adopted by the research community, to analyse a wide spectrum of scenarios. We have conducted code inspection and run different tests by hand to assure the correctness of these implementations. The source code has been checked by different individuals. The results obtained during these analyses are used to check if the MRs are fulfilled or not. Our evaluation of the MRs is based on the test manually generated by the user, that is, the source test cases. The follow-up test cases have been generated using random values and the corresponding constraints to assure the relation between the source test case and the generated one is fulfilled.

Other issues might arise due to the simulators used. These might have errors that can affect our findings. We have conducted experiments using different well-known simulators, which represent the behaviour of different scenarios of cloud systems to execute the tests. We mitigate

this threat during the experimental phase described in Section 8, where 20000 test cases were executed and checked over our proposed MRs.

9.2. External threats

External validity concerns the extent to which the results of a study can be generalised.

We have used three cloud configurations and three different workloads, inspired by big data analysis. Although we believe that these models are representative, there is no guarantee that the obtained results and the achieved improvements in the effectiveness of the MRs are the same for other scenarios.

Additionally, we found one case where the random approach provides a better result than the one generated by the proposed EA. Although it is expected that a guided search – using an EA – provides better results than a basic random approach, it is possible that, in some particular cases, the EA is not able to find a proper solution.

9.3. Constructs threats

Construct validity concerns whether the used measures are representative or not.

We measured the testing effectiveness of MRs based on the number of test cases that satisfy each MR, which is also widely used in the community. Defects in the simulators or in our proposed MRs could be a threat to construct validity, but we controlled this threat by executing a wide spectrum of test cases, using five cloud simulators to conduct our empirical study. After this experiment, we discarded three simulators because we detected some limitations, which do not properly represent the properties reflected in some MRs. Hence, we check that the MRs were properly designed and that our implementation worked correctly.

10. Conclusions and future work

In this paper, we have proposed `MT-EA4Cloud`, a methodology that combines EAs and MT to check the correctness of energy-aware cloud systems. In essence, this methodology is based on checking the satisfiability of MRs while testing cloud systems. To that end, we have proposed a complete catalogue of MRs that formally model the underlying infrastructure of cloud systems focusing on energy consumption.

In order to show the applicability of `MT-EA4Cloud`, we have performed an extensive experimental study, where three cloud models are analysed using seven well-known simulators and the provided catalogue of MRs. The experimental results obtained from this study are promising, demonstrating that it is feasible to combine EAs and MT to formally test cloud computing systems. Our proposed catalogue of MRs was used to check the correctness of different well-known simulators. Since each simulator provides specific capabilities to model the different parts of the cloud, our methodology can be applied to focus on those simulators that satisfy the user requirements. We also have observed that this approach can not only be used to analyse the correctness of simulation tools, but to discover flaws in cloud designs and to provide feasible solutions that improve these designs.

We can conclude that, during the testing process, the role of the expert is of vital importance. First, the expert is in charge of designing the MRs, providing source test cases and choosing an initial set of cloud simulators. Although there are steps in our methodology that can be automated, like the generation of a large number of follow-up test cases and the calculation of the MR effectiveness, her decisions have a direct impact of the final obtained results.

As future work, we plan to include heterogeneous cloud infrastructures, which provide machines exclusively dedicated to executing VMs (computing nodes) and machines dealing with the data accessed by the VMs (storage nodes). We are also interested in investigating the trade-off between cost and energy consumption. For this, a new EA should be designed dealing with the monetary cost of each component (e.g. CPUs, memories, networks) and to provide relevant information to the user about how the investment in better hardware impacts on the energy efficiency. Finally, we plan to study how dynamic workloads, which are generated in run-time, could be integrated into our framework. The main difficulty of this task lies in how these workloads are compared in the MRs.

Acknowledgments

This work was supported by the Spanish MINECO/FEDER projects DArDOS, FAME and MASSIVE under Grants TIN2015-65845-C3-1-R, RTI2018-093608-B-C31 and RTI2018-095255-B-I00, and the *Comunidad de Madrid* project FORTE-CM under grant S2018/TCS-4314. The first author is also supported by the Universidad Complutense de Madrid Santander Universidades grant (CT17/17-CT18/17).

[1] Ioannis Flouris and Nikos Giatrakos and Antonios Deligiannakis and Minos Garofalakis and Michael Kamp and Michael Mock, Issues in complex event processing: Status and prospects in the Big Data era, Journal of Systems and Software 127 (2017) 217 – 236.

[2] C.-F. Tsai, W.-C. Lin, S.-W. Ke, Big data mining with parallel computing: A comparison of distributed and MapReduce methodologies, Journal of Systems and Software 122 (2016) 83 – 92.

[3] K. Le, R. Bianchini, J. Zhang, Y. Jaluria, J. Meng, T. D. Nguyen, Reducing Electricity Cost Through Virtual Machine Placement in High Performance Computing Clouds, in: 24th Int. Conf. for High Performance Computing, Networking, Storage and Analysis, SC'11, ACM Press, 2011, pp. 22:1–22:12.

[4] Top500 Supercomputer sites, http://www.top500.org (April 2018) (2018).

[5] D.-M. Bui, Y. Yoon, E.-N. Huh, S. Jun, S. Lee, Energy efficiency for cloud computing system based on predictive optimization, Journal of Parallel and Distributed Computing 102 (2017) 103–114.

[6] Y. Sharma, B. Javadi, W. Si, D. Sun, Reliability and energy efficiency in cloud computing systems: Survey and taxonomy, Journal of Network and Computer Applications 74 (2016) 66–85.

[7] T. Chen, F. Kuo, H. Liu, P. Poon, D. Towey, T. Tse, Z. Zhou, Metamorphic testing: A review of challenges and opportunities, ACM Computing Surveys 51 (1) (2018) 4.

[8] S. Segura, G. Fraser, A. B. Sánchez, A. Ruiz-Cortés, A survey on metamorphic testing, IEEE Transactions on Software Engineering (in-press) PP (99) (2016) 1–1.

[9] K. A. D. Jong, Evolutionary Computation: A Unified Approach, march 25, 2016 Edition, A Bradford Book, 2016.

[10] M. Harman, Y. Jia, Y. Zhang, Achievements, open problems and challenges for search based software testing, in: IEEE 8th International Conference on Software Testing, Verification and Validation (ICST'15), 2015, pp. 1–12.

[11] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, R. Buyya, Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Software - Practice and Experience 41 (1) (2011) 23–50.

[12] G. Castañé, A. Núñez, P. Llopis, J. Carretero, E-mc^2: A formal framework for energy modelling in cloud computing, Simulation Modelling Practice and Theory 39 (2013) 56–75.

[13] H. Casanova, A. Legrand, M. Quinson, SimGrid: A generic framework for large-scale distributed experiments, in: 10th Int. Conf. on Computer Modeling and Simulation, UKSIM' 08, 2008, pp. 126–131.

[14] E. J. Weyuker, On testing non-testable programs, The Computer Journal 25 (4) (1982) 465–470.

[15] E. Gelenbe, Y. Caseau, The impact of information technology on energy consumption and carbon emissions, Ubiquity 2015 (June) (2015) 1:1–1:15.

[16] G. Pinto, F. Castor, Energy efficiency: a new concern for application software developers, Communications of the ACM 60 (12) (2017) 68–75.

[17] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, J. Clause, An empirical study of practitioners' perspectives on green software engineering, in: 38th International Conference on Software Engineering (ICSE'16), ACM, 2016, pp. 237–248.

[18] G. S. Akula, A. Potluri, Heuristics for migration with consolidation of ensembles of virtual machines, in: Sixth International Conference on Communication Systems and Networks (COMSNETS'14), 2014, pp. 1–4.

[19] S. F. Smith, Is scheduling a solved problem?, in: Multidisciplinary Scheduling: Theory and Applications (MISTA'03), 2005, pp. 3–17.
[20] S. Dillon, A. Quentin, M. Ivković, R. Furbank, E. Pinkard, Photosynthetic variation and responsiveness to co2 in a widespread riparian tree, PloS one 13 (1) (2018) e0189635.
[21] A. Khosravi, R. Buyya, Energy and carbon footprint-aware management of geo-distributed cloud data centers, Advancing cloud database systems and capacity planning with dynamic applications (2017) 27.
[22] Á. L. García, E. F. del Castillo, P. O. Fernández, I. C. Plasencia, J. Marco de Lucas, Resource provisioning in Science Clouds: Requirements and challenges, Software: Practice and Experience 48 (3) (2018) 486–498.
[23] J. Wang, K. Rao, H. Ye, Application-Specific, Performance-Aware Energy OptimizationUS Patent App. 15/224,834.
[24] P. Kurp, Green computing, Commun. ACM 51 (10) (2008) 11–13.
[25] The Green Grid, http://www.thegreengrid.org/ (April 2018) (2018).
[26] The Green 500 List, http://www.green500.org (2018).
[27] T. Y. Chen, S. C. Cheung, S. M. Yiu, Metamorphic testing: a new approach for generating next test cases, Tech. Rep. HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology (1998).
[28] J. Ding, D. Zhang, X. Hu, An application of metamorphic testing for testing scientific software, in: 1st International Workshop on Metamorphic Testing, ACM, 2016, pp. 37–43.
[29] H. Liu, F.-C. Kuo, D. Towey, T. Y. Chen, How effectively does metamorphic testing alleviate the oracle problem?, IEEE Transactions on Software Engineering 40 (1) (2014) 4–22.
[30] A. Beloglazov, J. Abawajy, R. Buyya, Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing, Future generation computer systems 28 (5) (2012) 755–768.
[31] I. Raïs, A. Orgerie, M. Quinson, L. Lefèvre, Quantifying the impact of shutdown techniques for energy-efficient data centers, Concurrency and Computation: Practice and Experience 30 (17).
[32] M. Sayadnavard, A. Haghighat, A. Rahmani, A reliable energy-aware approach for dynamic virtual machine consolidation in cloud data centers, The Journal of Supercomputing 75 (4) (2019) 2126–2147.
[33] M. Mohammadhosseini, A. Haghighat, E. Mahdipour, An efficient energy-aware method for virtual machine placement in cloud data centers using the cultural algorithm, The Journal of Supercomputing (2019) 1–30.
[34] M. Haghighi, M. Maeen, M. Haghparast, An Energy-Efficient Dynamic Resource Management Approach Based on Clustering and Meta-Heuristic Algorithms in Cloud Computing IaaS Platforms, Wireless Personal Communications 104 (4) (2019) 1367–1391.
[35] A. Merlo, M. Migliardi, L. Caviglione, A survey on energy-aware security mechanisms, Pervasive and Mobile Computing 24 (2015) 77–90.
[36] K. Sammy, R. Shengbing, C. Wilson, Energy efficient security preserving vm live migration in data centers for cloud computing, IJCSI International Journal of Computer Science Issues 9 (2) (2012) 1694–0814.
[37] V. Kharchenko, Y. Ponochovnyi, A. Boyarchuk, A. Gorbenko, Secure hybrid clouds: Analysis of configurations energy efficiency, in: International Conference on Dependability and Complex Systems, Springer, 2015, pp. 195–209.
[38] S. Segura, J. Parejo, J. Troya, A. Ruiz-Corts, Metamorphic Testing of RESTful Web APIs, IEEE Transactions on Software Engineering.
[39] M. Jiang, T. Y. Chen, F. Kuo, Z. Ding, Testing central processing unit scheduling algorithms using metamorphic testing, in: 4th IEEE International Conference on Software Engineering and Service Science, ICSESS'13, 2013, pp. 530–536.
[40] P. Rao, Z. Zheng, T. Y. Chen, N. Wang, K. Cai, Impacts of test suite's class imbalance on spectrum-based fault localization techniques, in: 13th International Conference on Quality Software (QSIC'13), 2013, pp. 260–267.
[41] X. Xie, W. E. Wong, T. Y. Chen, B. Xu, Metamorphic slice: An application in spectrum-based fault localization, Information and Software Technology 55 (5) (2013) 866–879.
[42] M. Hutchins, H. Foster, T. Goradia, T. Ostrand, Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria, in: 16th International Conference on Software Engineering (ICSE'94), 1994, pp. 191–200.
[43] V. Le, M. Afshari, Z. Su, Compiler validation via equivalence modulo inputs, in: 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14, ACM, 2014, pp. 216–226.
[44] W. K. Chan, T. Y. Chen, S. C. Cheung, T. H. Tse, Z. Zhang, Towards the testing of power-aware software applications for wireless sensor networks, in: Reliable Software Technologies – Ada Europe 2007, 2007, pp. 84–99.
[45] A. Núñez, R. M. Hierons, A methodology for validating cloud models using metamorphic testing, Annales of Telecommunications 70 (3-4) (2015) 127–135.
[46] C. Murphy, M. S. Raunak, A. King, S. Chen, C. Imbriano, G. Kaiser, I. Lee, O. Sokolsky, L. Clarke, L. Osterweil, On effective testing of health care simulation software, in: 3rd Workshop on Software Engineering in Health Care, SEHC '11, 2011, pp. 40–47.
[47] J. Ding, T. Wu, D. Wu, J. Q. Lu, X. Hu, Metamorphic testing of a monte carlo modeling program, in: 6th Interna-

tional Workshop on Automation of Software Test, AST '11, 2011, pp. 1–7.
[48] T. Y. Chen, F. Kuo, H. Liu, S. Wang, Conformance testing of network simulators based on metamorphic testing technique, in: 11th IFIP WG 6.1 International Conference FMOODS '09, 2009, pp. 243–248.
[49] T. Chen, F. C. Kuo, R. Merkel, W. K. Tam, Testing an open source suite for open queuing network modelling using metamorphic testing technique, in: 14th IEEE International Conference on Engineering of Complex Computer Systems, 2009, pp. 23–29.
[50] I. Zelinka, A survey on evolutionary algorithms dynamics and its complexity mutual relations, past, present and future, Swarm and Evolutionary Computation 25 (2015) 2 – 14.
[51] B. Keshanchi, A. Souri, N. J. Navimipour, An improved genetic algorithm for task scheduling in the cloud environments using the priority queues: formal verification, simulation, and statistical testing, Journal of Systems and Software 124 (2017) 1–21.
[52] M. Vasudevan, Y.-C. Tian, M. Tang, E. Kozan, X. Zhang, Energy-efficient application assignment in profile-based data center management through a repairing genetic algorithm, Applied Soft Computing 67 (2018) 399–408.
[53] Z. Xiao, J. Jiang, Y. Zhu, Z. Ming, S. Zhong, S. Cai, A solution of dynamic vms placement problem for energy consumption optimization based on evolutionary game theory, Journal of Systems and Software 101 (C) (2015) 260–272.
[54] H. Ibrahim, R. Aburukba, K. El-Fakih, An integer linear programming model and adaptive genetic algorithm approach to minimize energy consumption of cloud computing data centers, Computers & Electrical Engineering 67 (2018) 551–565.
[55] S. Segura, J. Troya, A. Durán, A. Ruiz-Cortés, Performance metamorphic testing: A proof of concept, Information and Software Technology 98 (2018) 1–4.
[56] J. Rounds, U. Kanewala, Systematic testing of genetic algorithms: A metamorphic testing based approach, arXiv preprint arXiv:1808.01033.
[57] D. Arora, V. G. Bassi, Generating test cases using metamorphic testing and genetic algorithm for integer bugs detection, Ph.D. thesis (2015).
[58] E. Elbeltagi, T. Hegazy, D. Grierson, Comparison among five evolutionary-based optimization algorithms, Advanced engineering informatics 19 (1) (2005) 43–53.
[59] V. Kachitvichyanukul, Comparison of three evolutionary algorithms: GA, PSO, and DE, Industrial Engineering and Management Systems 11 (3) (2012) 215–223.
[60] J. Wegener, K. Grimm, M. Grochtmann, H. Sthamer, B. Jones, Systematic testing of real-time systems, in: 4th International Conference on Software Testing Analysis and Review (EuroSTAR 96), 1996.
[61] K. Wloch, P. Bentley, Optimising the performance of a formula one car using a genetic algorithm, in: International Conference on Parallel Problem Solving from Nature, Springer, 2004, pp. 702–711.
[62] J. Byrne, S. Svorobej, K. Giannoutakis, D. Tzovaras, P. Byrne, P. Ostberg, A. Gourinovitch, T. Lynn, A review of cloud computing simulation platforms and related environments, in: 7th International Conference on Cloud Computing and Services Science, 2017, pp. 679–691.
[63] M. Tighe, G. Keller, M. Bauer, H. Lutfiyya, DCSim: A data centre simulation tool for evaluating dynamic virtualized resource management, in: 8th International conference on network and service management, 2012, pp. 385–392.
[64] S. U. K. Dzmitry Kliazovich, Pascal Bouvry, GreenCloud: A packet-level simulator of energy-aware cloud computing data centers, The Journal of Supercomputing 62 (3) (2012) 1263–1283.
[65] A. Núñez, J. L. Vázquez-Poletti, A. C. Caminero, G. G. Castañé, J. Carretero, I. M. Llorente, iCanCloud: A flexible and scalable cloud infrastructure simulator, Journal of Grid Computing 10 (1) (2012) 185–209.
[66] G. Kecskemeti, DISSECT-CF: A simulator to foster energy-aware scheduling in infrastructure clouds, Simulation Modelling Practice and Theory 58 (2015) 188–218, special issue on Cloud Simulation.
[67] H. Ouarnoughi, J. Boukhobza, F. Singhoff, S. Rubini, Integrating i/os in cloudsim for performance and energy estimation, ACM SIGOPS Operating Systems Review 50 (1) (2017) 27–36.
[68] P. Cristian, P. Eugen, A. Marcel, C. Pop, T. Cioara, I. Anghel, I. Salomie, Coolcloudsim: Integrating cooling system models in cloudsim, in: IEEE 14th International Conference on Intelligent Computer Communication and Processing (ICCP'18), 2018, pp. 387–394.
[69] A. S. M. Rizvi, T. R. Toha, M. M. R. Lunar, M. A. Adnan, A. B. M. A. A. Islam, Cooling energy integration in simgrid, in: 2017 International Conference on Networking, Systems and Security (NSysS), 2017, pp. 132–137.
[70] K. Kurowski and A. Oleksiak and W. Piatek and T. Piontek and A. Przybyszewski and J. Weglarz, Dcworms - a tool for simulation of energy efficiency in distributed computing infrastructures, Simulation Modelling Practice and Theory 39 (2013) 135–151.
[71] N. Hassan, M. Khan, M. Rasul, Temperature monitoring and cfd analysis of data centre, Procedia Engineering 56 (2013) 551 – 559, 5th BSME International Conference on Thermal Engineering.
[72] D. Ortiz-Boyer, C. Hervás-Martínez, N. García-Pedrajas, Cixl2: a crossover operator for evolutionary algorithms based on population features, Journal of Artificial Intelligence Research 24 (2005) 1–48.

[73] A. Globus, S. Atsatt, J. Lawton, T. Wipke, Javagenes: Evolving graphs with crossover, Tech. Rep. NAS-00-006, NASA Advanced Supercomputing Division (2000).
[74] K. Park, V. S. Pai, CoMon: a mostly-scalable monitoring system for PlanetLab, ACM SIGOPS Operating Systems Review 40 (1) (2006) 65–74.
[75] W. Kolberg, P. B. Marcos, J. C. Anjos, A. Miyazaki, C. Geyer, L. Arantes, Mrsg–a mapreduce simulator over simGrid, Parallel Computing 39 (4-5) (2013) 233–244.

7.3 MuTomVo: mutation testing framework for simulated cloud and HPC environments

7.3	
Publication data	
Authors:	Pablo C. Cañizares, Alberto Núñez and Mercedes G. Merayo
Title:	MuTomVo: mutation testing framework for simulated cloud and HPC environments
Publication:	Journal of Systems and Software
Pub. Type:	Journal
Year:	2018
DOI/URL:	https://doi.org/10.1016/j.jss.2018.05.010
Pages:	21
Category:	Computer Science, Software Engineering
Quartile:	Q1
Ranking:	26/107
Impact factor:	2.559
Contribution	
Summary:	In this paper we propose a mutation testing framework for detecting errors in distributed applications executed in simulated cloud and HPC environments. The execution of a test suite against the set of mutated models allows to determine its effectiveness for detecting different errors. The proposal has been implemented in a tool called MuTomVo. In order to support the feasibility of the proposal, a case study has been conducted using three applications running in different distributed systems: a client/server model, intensive computation and scientific pipeline.
Technique:	Mutation Testing.
Secondary techniques:	Simulation

The Journal of Systems & Software 143 (2018) 187–207

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Controversy Corner

Mutomvo: Mutation testing framework for simulated cloud and HPC environments

Pablo C. Cañizares, Alberto Núñez*, Mercedes G. Merayo

Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Spain

ARTICLE INFO

Keywords:
Software testing
Mutation testing
Distributed systems
Simulation

ABSTRACT

Many current applications provide high performance to process large volumes of data. These applications usually run in highly distributed environments, like cloud and HPC systems. Nevertheless, the large and complex architectures required for deploying these applications may not be available during the development phase. This limitation can be overcome by using simulation platforms to model a wide range of distributed system configurations and execute these applications in the modeled system. Usually, these applications are tested against a small number of test cases that are manually designed by the testers. It is desirable to have effective test suites in order to detect failures in the application models. In this paper we propose a mutation testing framework for detecting errors in distributed applications executed in simulated environments. The execution of a test suite against the set of mutated models allows to determine its effectiveness for detecting different errors. The proposal has been implemented in a tool called MuTomVo. In order to support the feasibility of the proposal, we have carried out a case study over three applications running in different distributed systems: a client/server model, intensive computation and scientific pipeline.

1. Introduction

Over the last decades, there has been a change in business models as a consequence of the high-speed Internet era. This new model carries a growth of the infrastructure and services, increasing both data generation and processing (Gros and Alcidi, 2013). This revolution requires the development and deployment of new applications that provide high performance to process large volumes of data. Generally, these applications are executed in distributed systems that provide high performance resources, like cloud and HPC systems (Schad, 2010; Althebyan et al., 2015; Ried et al., 2011).

However, processing large volumes of data increases the complexity of the applications that, usually, must be deployed in large and complex architectures that may not be available during the development process. For this reason, in order to overcome the previously described difficulties, the research community has opted for the use of techniques that do not require real resources and, in addition, reduce the high cost associated with them. This is the case of the simulation techniques. Currently, there exist several tools for simulating, from individual components such as disks (Bucy et al., 2008) or CPUs (Liu and Fan, 2011) to complete cloud systems (Buyya and Calheiros, 2009; Castañé et al., 2012). These tools allow to model and represent the behavior of highly distributed architectures, where applications are executed.

Some advantages obtained by the use of simulation tools to execute applications, in contrast with executing applications in the physical system are:

- Flexibility to configure the system architecture in which experiments are performed. Making changes in data centers can be very costly because they require hardware modifications, such as, among others, the topology of the network. However, simulation tools allow to make changes in simulated architectures in a simple and flexible way.
- Reproducing experiments in a controlled environment. In the simulation, there are no external factors that may affect the result of the tests. Both virtualization, in the case of cloud systems, and the concurrent execution of multiple users do not impact on simulation. In this project we have used deterministic simulation, that is, if a model is simulated several times, the same result is obtained.
- Accessing to infrastructure in which experiments are performed. Due to the high cost of these systems, they are only available to large corporations or institutions. Using simulation tools allows to perform experiments on modeled architectures without purchasing specific hardware.
- Investing is required to access resources. The use of resources in cloud systems requires capital investment. The users, through a

* Corresponding author.
E-mail addresses: pablocc@ucm.es (P.C. Cañizares), alberto.nunez@pdi.ucm.es (A. Núñez), mgmerayo@fdi.ucm.es (M.G. Merayo).

https://doi.org/10.1016/j.jss.2018.05.010
Received 1 December 2017; Received in revised form 30 April 2018; Accepted 7 May 2018
Available online 03 June 2018

service provider, select an adequate computing resources to cover their needs and rent them for a period of time. The cost can be avoided by modeling these hardware resources to be used in a modeling environment.

Generally, simulation platforms offer APIs that provide a catalogue of functions for accessing the different resources of the system, such as disks, CPUs and communication networks. Basically, the idea is to write applications using the provided APIs and, once the target system is modeled, executing–using a simulator–these applications on the simulated architecture. In order to ensure that these applications fulfill the expected behavior, the tester should design and execute a proper test suite over the application under test. To that end, simulation allows to test distributed applications when the system infrastructure is expensive and not available.

Unfortunately, in most cases testing techniques are not applied. Instead, the application is checked using a small test suite that, in general, the tester manually generates. Thus, a vast number of unexplored potential errors are not detected. This lack of *appropriate* test suites raises the possibility of failure. Fortunately, there exist several techniques to address this issue. One of them is known as *mutation testing* (DeMillo et al., 1978). Basically, mutation testing introduces syntactic changes in the source code of a program by generating several versions of it, each containing a fault. These faulty programs, called *mutants*, are executed against the test suite with the goal of determining its effectiveness in finding errors. In this paper, we propose the use of mutation testing techniques in simulation environments. In order to reach this objective, we define a set of *mutation operators* that will be used to create mutants of applications designed to be executed in simulated distributed systems. We have implemented the mutation testing process in a tool called MuTomVo, designed to be used in simulation tools based on OMNeT++ (Varga, 2001). The simulation platform that we have chosen for developing our framework is SIMCAN[1] (Núñez et al., 2012) due to the fact that we know it in depth. Nevertheless, MuTomVo can be used with any other simulation tool based in OMNeT++. In summary, we use SIMCAN to *simulate* different distributed architectures, where applications are executed to provide an output. These applications are *tested* using test suites. The correctness of a test suite is checked by applying mutation testing techniques. Thus, applications are *mutated* by injecting faults in its source code.

In order to check the feasibility of MuTomVo we have carried out a case study in which the mutation process has been performed over three applications executed in different distributed systems: a client/server model, intensive computation and scientific pipeline. Since we think that traditional mutation operators are not adequate for testing simulated distributed applications, we also compare the effectiveness achieved by these operators with the effectiveness obtained when our proposed operators are applied.

The specific contributions of the proposed framework described in this paper are:

- Applying mutation testing in highly scalable distributed systems, efficiently and accurately.
- Designing new mutation operators focused on simulation. These operators represent different errors made by competent programmers over these platforms.
- Developing a specific tool that implements the proposed mutation testing framework in a simulation tool.
- Evaluating the proposed framework to determine the applicability of the proposed framework and analyze the effectiveness of test suites for detecting errors in distributed applications.

The rest of the paper is structured as follows. Section 2 provides an overview of SIMCAN, the simulation platform for modeling and simulating both distributed systems and applications. In Section 3 we detail the mutation operators that will be applied in our framework. In Section 4 the main features of MuTomVo are explained. Section 5 presents the performed experiments and an analysis of the obtained results. The threats to validity of our experiments are explained in Section 6. Section 7 review related approaches. Finally, in Section 8 we present our conclusions and future work.

2. The SIMCAN simulation platform

In this section we present an overview of SIMCAN (Núñez et al., 2012), a simulation platform for modeling and simulating both distributed systems and applications, which is currently available as open source software. SIMCAN has been written in C++ using the OMNeT++ (Varga, 2001), a simulation framework focused on building network simulators. Currently, OMNeT++ is one of the most extended and active simulation frameworks in the scientific community, having been used in more than 2000 scientific publications during the last five years[2].

During the last decade, simulation has become one of the most used options to carry out scientific experiments (Winsberg, 2010). In the field of distributed systems, simulation techniques are especially useful when large and complex architectures are not available. Hence, SIMCAN has been designed to fulfill three objectives. First, to provide a high level of flexibility and scalability, allowing users to model a wide range of distributed system configurations. Second, to ease the development of distributed applications by providing intuitive and easy-to-use APIs. Third, to execute these applications in the modeled distributed systems.

The simulation core of SIMCAN relies on its repository. Basically, this repository contains a collection of models that represents the most relevant components of a distributed system, such as CPUs, disks and communication networks. These models are hierarchically classified into four basic systems: storage, CPU, memory and network. The same component can be represented by different models, e.g. a CPU processor can be modeled as a Single-Core CPU model and as a Quad-Core CPU model. Moreover, new models can be included in the repository, increasing the number of system configurations that can be built by combining the existent models.

Using this structure, users are able to model large distributed systems, like HPC clusters and data-centers for supporting cloud computing environments. In this case, several computer models are interconnected through a communication network. SIMCAN provides the same aggregation method as the one used in real distributed systems, that is, a rack structure contains several node boards, where each board contains several computers. The size of these racks and node boards is fully configurable. Hence, large systems can be easily deployed by using this aggregation structure. Moreover, a GUI written in Java is provided (see Fig. 1). The main goal of this application is two-fold. First, hiding all low-level details, including the language used for configuring hardware components. Second, easing the modeling of large distributed system by browsing and loading the core modules.

In SIMCAN, a computer can be modeled by defining the four basic systems that correspond to the different components that can be found in the repository. Each computer model contains an API module that connects the applications with the four basic systems. Thus, user applications are able to request hardware resources by invoking the functions provided by the API module.

Simulated applications in SIMCAN do not have to deal with real data, but using a statistical approach to representing the behavior of the application. Therefore, each model that simulates a hardware component provides an estimation of the real time required for using the

[1] Available at http://www.simcansimulator.com.

[2] Based on Google Scholar search results.

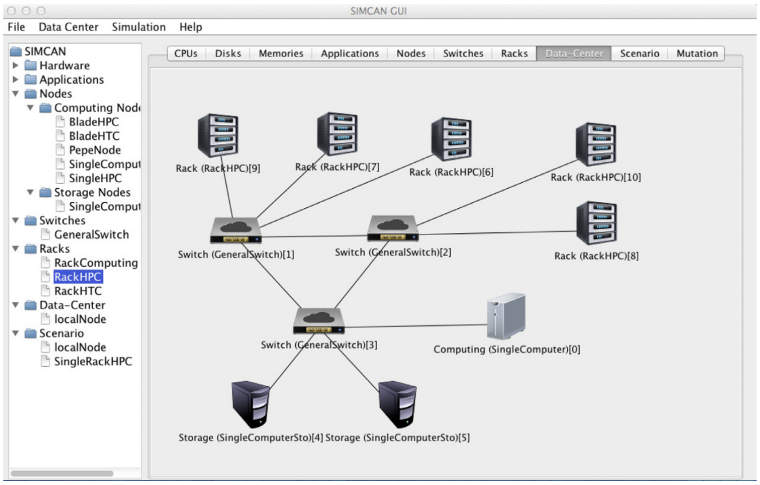


Fig. 1. Screenshot of the GUI provided with SIMCAN.

corresponding service. The key to obtaining a good application is to balance the level of abstraction of the application's characteristics and the level of computation required to execute the simulation. However, this balance must be calculated and designed by the programmer that writes the application to be simulated, which in some cases it becomes a hard and complex task for inexperienced users.

Following subsections describe in detail the different APIs used in this work.

2.1. OMNeT++ API

The most common methods included in the OMNeT++ API that are used to develop new modules are presented in Listing 1.

The *initialize* and *finish* methods are called at the beginning and the end of the simulation, respectively. The first one may include the initialization code. The second one only will be called if the simulation has finished successfully. The methods *send* and *scheduleAt* allow to send messages. The former sends a message *msg* to other modules via a specific gate *gateid*. The later schedules a self-message that will be delivered back to the module at simulation time *t*. This method is very useful for implementing timers or delays. The method *cancelEvent* can be used to cancel the self-message before it arrives and *cancelAndDelete* invokes *cancelEvent* on the message and deletes it. The *endSimulation* method stops the whole simulation process. Finally, the messages among modules are managed via the *handleMessage* method. Generally, this is the most important method in a module because it contains the code that defines its behavior. Due to the fact that *initialize, finish* and *handleMessage* are virtual member functions, they must be redefined by the user to add functionality to the module but they are not called directly, therefore, we do not consider them in the definition of the mutation operators.

2.2. SIMCAN API

In SIMCAN, the API module provides an interface that has an essential role in the component that simulates the operating system. Basically, this interface provides a set of functions that are invoked by the user applications to request resources to the operating system. The functions included in the SIMCAN API appear in Listing 2.

The function *simcan_request_cpu* allows applications to request CPU resources, based on the number of millons of instructions that must be executed. In order to allocate and free memory, the applications must invoke the *simcan_request_allocMemory* and *simcan_request_freeMemory*, respectively. The set of functions shown at lines $5-8$ provides an interface to interact with the storage system. Basically, these functions (*simcan_request_open, simcan_request_close, simcan_request_create, simcan_request_delete*) are used to manage storage files. The functions *simcan_request_read* and *simcan_request_write* allow to perform read and write operations.

The management of connections to interchange data with other applications running on remote nodes, must be done by invoking the functions *simcan_request_createListenConnection* and *simcan_request_createConnection*. The former one allows to create an incoming connection and the latter one establishes a connection with a remote application. Finally, the functions *simcan_request_sendDataToNetwork* and *simcan_request_receiveDataFromNetwork* are used to send/receive data to/from a specific remote application.

2.3. MPI API

In addition to the previously described functions, SIMCAN also provides a subset of the functions included in the MPI library. MPI is used for programming parallel applications in distributed architectures. The MPI calls implemented in SIMCAN, shown in Listing 3, can be

```
1 void initialize()
2 void finish()
3 int send(cMessage* msg, int gateid)
4 int scheduleAt(simtime_t t, cMessage* msg)
5 cMessage* cancelEvent(cMessage* msg)
6 void cancelAndDelete(cMessage* msg)
7 void endSimulation()
8 void handleMessage(cMessage* msg)
```

Listing 1. OMNeT++ Methods.

```
void simcan_request_cpu (long int MIs)
void simcan_request_cpuTime (simtime_t cpuTime)
void simcan_request_allocMemory (int memorySize, int region)
void simcan_request_freeMemory (int memorySize, int region)
void simcan_request_open (char* file)
void simcan_request_close (char* file)
void simcan_request_create (char* file)
void simcan_request_delete (char* file)
void* simcan_request_read (char* file, int offset, unsigned int size)
void* simcan_request_write (char* file, int offset, unsigned int size)
void simcan_request_createListenConnection (int localPort, string type)
void simcan_request_createConnection (string host, int port, int id,
    string t)
void simcan_request_sendDataToNetwork (simcan_Message* sm, int id)
void simcan_request_receiveDataFromNetwork (simcan_Message* sm, int id)
```

Listing 2. API functions offered by SIMCAN.

```
void mpi_send(unsigned int rankReceiver, int bufferSize)
void mpi_send(simcan_MPI_Message* sm)
simcan_MPI_Message* mpi_recv(unsigned int rankSender, int bufferSize)
void mpi_barrier()
void mpi_bcast(unsigned int root, int bufferSize)
void mpi_scatter(unsigned int root, int bufferSize)
void mpi_gather(unsigned int root, int bufferSize)
```

Listing 3. MPI collective functions implemented in SIMCAN.

classified in two different groups. The first group includes point to point communication operations: *mpi_send* and *mpi_recv* associated with sending and receiving data, respectively. The second group corresponds to collective calls. These calls provide synchronization and operations among processes that belong to a communicator. A communicator encompasses a group of processes that may communicate with each other. The function *mpi_barrier* is used to synchronize processes in a communicator. When a process invokes this function, this process is blocked until all processes in the communicator have called it. In order to send a message from a *root* process to the rest of the processes in a communicator, the *mpi_bcast* must be invoked. The function *mpi_scatter* also allows to send data to all processes in a communicator. The main difference between *mpi_bcast* and *mpi_scatter* is that *mpi_bcast* sends the same piece of data to all processes while *mpi_scatter* sends chunks of data to different processes. Finally, *mpi_gather* collects data from different processes to a *root* process.

2.4. Modeling with SIMCAN

This section presents an example that illustrates how to code in SIMCAN a C++ existing application that uses a traditional structured paradigm. Basically, this application (see Listing 4) reads two matrixes from disk, calculates the multiplication of these matrixes and stores the results in the disk. The function *readMatrix* loads a matrix stored in disk into memory. Similarly, the function *writeMatrixToFile* stores in disk the matrix that is allocated in memory. Finally, the function *multMatrixes* calculates the product of two matrixes.

This application, which has been written for SIMCAN, (see Listing 5) is based on events. It is important to mention that the SIMCAN simulation platform entails an important limitation for creating new applications. Since SIMCAN has been built over the OMNET++ platform, each application must be programmed by using the event-oriented programming paradigm, which is more complicated than using a traditional structured paradigm. For each function from the API that is invoked, the simulator calculates the amount of time required to process it. Once the time for this request has elapsed, the response is sent to the application. In order to invoke these functions in the correct order, we use a state sequence in the application.

In the first two states, the method *SIMCAN_request_read* is invoked to read the two input matrixes, $m1$ and $m2$, from disk. Next, once the application reaches the third state, called *READM2*, the file *m3*, that will contain the result, is created by using the method *SIMCAN_request_create*. In the next state, called *CREATM3*,

```
void main(){

int fd1, fd2, fd3;
void * m1, m2, m3;

  // Open and create files
  fd1=open("m1.txt", O_RDONLY);
  fd2=open("m2.txt", O_RDONLY);
  fd3=creat("m3.txt", O_WRONLY);

  // Read matrixes
  m1 = readMatrix (fd1);
  m2 = readMatrix (fd2);

  // Multiplicate m1 x m2
  m3 = multMatrixes (m1, m2);

  // Write result
  writeMatrixToFile (m3, fd3);


}
```

Listing 4. C++ structured application.

calculateTime estimates, based on the size of the two input matrixes, the required computing time to perform the multiplication of these matrixes. This computing time is stored in $t3$, which is passed, as an input parameter, to *SIMCAN_request_cpuTime* for simulating the calculation of the product of $m1$ and $m2$. Finally, the last state of this application writes the result in the file *m3.txt* by invoking *SIMCAN_request_create*.

In the next sections we introduce the mutation operators that represent common programmer errors during the implementation of applications for simulation. The proposed operators are intended to replicate unexpected behaviors of the application that may happen during the execution, such as communication errors and deadlocks. Following, we describe the structure of the test cases that will be used in order to detect the errors injected by the mutation operators. A test case corresponds to a set of pairs that configures the execution of the application and its mutants.

The Listing 6 shows a test case for this application. It contains all the necessary parameters for the simulation of the application. The values

```
void processEvent (Message *msg) {
switch (state) {
case INIT:
  SIMCAN_request_read("m1.txt");
  m1 = msg->getSize();
  state=READ_M1; return(0);
case READ_M1:
  SIMCAN_request_read("m2.txt", 0,
       sizeof(m2));
  m2 = msg->getSize();
  state=READ_M2; return(0);
case READ_M2:
   SIMCAN_request_create("m3.txt");
   state=CREAT_M3; return(0);
case CREAT_M3:
   t3 = calculateTime(m1, m2);
   SIMCAN_request_cpuTime(t3);
   state=WRITE_M3; return(0);
case WRITE_M3:
   SIMCAN_request_write("m3.txt", 0,
        sizeof(m1));
   state=END; return(0);
case END:
   endSimulation();
 }
```

Listing 5. SIMCAN application of Listing 4.

returned by the execution of the application against the previous test case will be used as the oracle to check the correct/incorrect behavior of the faulty versions.

3. Mutation operators based on the API of distributed applications

This section provides a detailed description of the proposed mutation operators for distributed applications. We designed 23 mutation operators, which were derived from a list of common errors extracted from different sources. These operators focus on the functions used by distributed applications in simulated environments that are provided by OMNeT++, SIMCAN and MPI APIs. The main objective of these operators is to generate a reduced number of mutants that have a significant impact in the source code of the applications under test. The design of these operators is inspired by the principles of the cost reduction techniques for mutation testing: "do fewer, do faster and do smarter" (Jia and Harman, 2011). Since good mutation operators provide significant mutants representing typical errors produced by competent programers, poor mutation operators may produce the opposite, that is, a large number of redundant mutants. Hence, using these operators we expect to obtain better results and performance than using traditional mutation operators, such as Arithmetic Operator Replacement (AOR) and Logical Operator Insertion (LOI) (Offutt and King, 1987; Ammann and Offutt, 2008).

The process for designing these operators has been developed in two phases. In the first phase, we used the expertise of two advanced programmers focused on simulation models, to create a list of the most common mistakes when a distributed application is written to be executed in a simulated environment. Some of these errors were gathered during the validation process of the SIMCAN simulation platform (Núñez, 2011). A Ph.D. student extended the list by using all the detected errors during the development of different SIMCAN applications. Table 1 provides the list of common errors found during this phase along with the derived mutation operators.

During the second phase we analyzed different sources to gather existing errors in real applications. First, different repositories containing OMNeT++ applications were studied. In particular, we focus our efforts in analyzing bug reports, mailing lists and "Whats new" logs from the INET (Szaszko, 2017), overSim (Furnaghan, 2014) and RINAsim (Vesely, 2017) simulators, among others. The idea of investigating bug reports and history logs for searching common errors produced by real programmers is not new. Deng et. al use common faults found in different source code repositories to design mutation operators (Deng et al., 2017). This analysis allowed us to identify common errors existing in different applications, like wrong timing set in *scheduleAt* calls or parameters from ned modules that were not read with the *par* function. We have also reviewed works from the literature, focusing on analyzing common errors produced by programmers. DeSouza et al. (2005) identified a list of common errors produced by programmers of MPI applications. Since this list focuses on real MPI applications, we only used those errors that can be applied to simulation environments. For example, we are able to identify errors based on send/receive inconsistencies (i. e. using a wrong process ID) but we cannot detect errors produced after changes in the compilation environment, which are produced in real environments using compilers and MPI libraries. From the best of our knowledge, there is no public repository of SIMCAN applications and, therefore, we have generated a list of errors inspired by the some MPI errors produced to interchange data between processes. For instance, while send/receive MPI calls

```
sm->removeLastModuleFromTrace ()
    ;
send (sm, gateId); ◀
if(updateSentMessages()>=MAX_MSG
    )
{
    endSimulation();
}
```

Listing 7. Original code.

```
sm->removeLastModuleFromTrace ()
    ;
◀
if(updateSentMessages()>=MAX_MSG
    )
{
    endSimulation();
}
```

Listing 8. OMCD Equivalent mutant.

```
inputSize  = (int)
    par ("inputSize") * MB;
outputSize  = (int)
    par ("outputSize") * MB; ◀
MIs  = par ("MIs");
```

Listing 9. Original code.

```
inputSize  = (int)
    par ("inputSize") * MB;
                          ◀

MIs  = par ("MIs");
```

Listing 10. OOPD mutant.

```
int M1Rows=2000;          // Rows of matrix 1
int M1Columns=1500;       // Columns of matrix 1
int M1Rows=3000;          // Rows of matrix 2
int M1Columns=1000;       // Columns of matrix 2
int MIs=45896;            // Computing for each iteration
```

Listing 6. Example of test case.

Table 1
List of errors found in phase 1 and the derived mutation operators.

ID_ERROR	Description	Operator
ISSUE#001# [OMN_CALL_DEL]	Missing OMNET++ method call.	OMCD - OMNeT++ method call deletion.
ISSUE#002# [OMN_CALL_REP]	Mismatch while invoking an OMNET++ method call.	OMCR - OMNeT++ method call replacement.
ISSUE#003# [OMN_MV_UP]	Wrong order of OMNET++ calls in the source code.	OOMU - OMNeT++ operator move up.
ISSUE#004# [OMN_MV_DOWN]	Wrong order of OMNET++ calls in the source code.	OOMD - OMNeT++ operator move down.
ISSUE#005# [SIM_CALL_DEL]	Missing SIMCAN method call.	SMCD - SIMCAN method call deletion.
ISSUE#006# [SIM_CALL_REP]	Mismatch while invoking a SIMCAN method call.	SMCR - SIMCAN method call replacement.
ISSUE#007# [SIM_MV_UP]	Wrong order of a SIMCAN call in the source code.	SOMU - SIMCAN operator move up.
ISSUE#008# [SIM_MV_DOWN]	Wrong order of a SIMCAN call in the source code.	SOMD - SIMCAN operatormove down.
ISSUE#009# [SIM_SHF_PRM]	Wrong order of the parameters in SIMCAN method calls.	SOSP - SIMCAN operator shuffle parameters.
ISSUE#010# [MPI_CALL_DEL]	Missing MPI method call.	MMCD - MPI method call deletion.
ISSUE#011# [MPI_CALL_REP]	Mismatch while invoking an MPI method call.	MMCR - MPI method call replacement.
ISSUE#012# [MPI_MV_UP]	Wrong order of an MPI call in the source code.	MOMU - MPI operator move up.
ISSUE#013# [MPI_MV_DOWN]	Wrong order of an MPI call in the source code.	MOMD - MPI operator move down.
ISSUE#014# [MPI_SHF_PRM]	Wrong order of the parameters in MPI method calls.	MOSP - MPI operator shuffling parameters.

Table 2
List of errors found in phase 2 and the derived mutation operators.

ID_ERROR	Description	Operator
ISSUE#015# [OMN_PAR_DEL]	Missing par call.	OOPD - OMNeT++ operator par deletion.
ISSUE#016# [OMN_SIG_SWAP]	Signals swapped.	OOSS - OMNeT++ operator signal swapping.
ISSUE#017# [OMN_SIG_N_REP]	Error while using an undefined signal. (Mismatching signal name)	OOSNR - OMNeT++ operator signal name replacement.
ISSUE#018# [OMN_TIME_REP]	Issue on scheduling time.	OOSTR - OMNeT++ operator scheduling time replacement.
ISSUE#019# [SIM_FILE_N_REP]	Error while using an undefined file name. (Mismatching name)	SOFNR - SIMCAN operator file name replacement.
ISSUE#020# [SIM_MALLOC_REP]	Wrong memory allocation.	SOMAR - SIMCAN operator memory allocation replacement.
ISSUE#021# [SIM_COM_ID_REP]	Mismatching destination ID in communication operations.	SOCIR - SIMCAN operator communication ID replacement.
ISSUE#022# [MPI_PROC_ID_REP]	Mismatching process destination ID in MPI operations.	MOPIR - MPI operator process ID replacement.
ISSUE#023# [MPI_BUF_LEN_REP]	Error in the buffer length using MPI operations.	MOBLR - MPI operator buffer length replacement.

must specify the sender/receiver process ID, in SIMCAN, the programmer must specify the communication ID obtained when such communication is established (see Listing 2). Hence, these errors can be similarly generated by using operators focused on the SIMCAN API. Other kind of errors, like those related to the memory management, were gathered from the works developed by Nanavati et al. (2015) and Wu et al. (2017). The list of errors found in this phase and the corresponding derived operators are shown in Table 2.

3.1. Deletion operators

The mutation operators included in this category remove calls to methods included in OMNeT++, SIMCAN and MPI APIs. Although, in most cases, the consequence of removing a call consists on a reduction of the CPU or memory usage, there are some cases in which the deletion of a sentence can have different effects on the behavior of the system. We introduce four different remove operators, *OMCD, OOPD, SMCD* and *MMCD*, that can be applied to calls to methods included in the OMNET++, SIMCAN and MPI APIs, respectively.

3.1.1. OMCD - OMNET++ method call deletion

The *OMCD* operator removes calls to the only methods in OMNET++ API that can be directly called, that is, *cancelEvent, EndSimulation, send* and *scheduleAt*. Removing the *cancelEvent* method can lead to an infinite loop. If the *EndSimulation* is removed, the simulation does not stop. In the case of *send*, messages are not sent and if a *scheduleAt* call is removed the system might stay waiting for an event indefinitely. Next, we present an example of the *OMCD* operator used to remove a *send* method call.

3.1.2. OOPD - OMNET++ operator par deletion

This operator removes the *par* call from the source code, which obtains the value of a parameter from the text files that configure the simulation environment. This method is provided by the OMNeT++ API. The following example shows a mutant that deletes one *par* call, keeping non-initialized the *outputSize* parameter.

3.1.3. SMCD - SIMCAN method call deletion

The *SMCD* operator removes calls to any function included in the SIMCAN API. Removing *simcan_request_cpu* or *simcan_request_cpuTime* affects the CPU usage, while removing *simcan_request_allocMemory* or *simcan_request_freeMemory* modifies the memory usage. When a call to the *simcan_request_read* or *simcan_request_write* is removed, the amount of data that are read or written to disk decreases. The removal of a call to the *simcan_request_sendDataToNetwork* or *simcan_request_receiveDataFromNetwork* methods might influence the network resource consumption. Finally, the consequence of deleting a call to the *simcan_request_createConnection* or the *simcan_request_createListenConnection* methods makes impossible to send/receive data through the corresponding channel. Next example shows a mutant obtained by the application of the *SMCD* operator to a call to the *simcan_request_cpu* method.

3.1.4. MMCD - MPI method call deletion

Similar to the *SMCD* operator, the *MMCD* removes calls to any function in the MPI API. In the case of deletion of a call to *mpi_barrier* causes a deadlock, because it is used to synchronize processes in a communicator. Removing a call to any of the other functions included in the API affects the network resource consumption. The following example presents a mutant obtained by the application of the *MMCD* operator to *mpi_send*.

3.2. Method replacement operators

This kind of operators is focused on the replacement of a method call with a different one that presents the same number and type of parameters. In the following we detail the replacements that can be applied by the replace operators *OMCR, SMCR* and *MMCR*, to methods included in the OMNET++, SIMCAN and MPI APIs, respectively.

```
simcan_request_cpu(calcTime(
    procCPU)); ◀
scheduleAt(simTime()+timeETP,
    timeOutMsg);
```

Listing 11. Original code.

```
◀
scheduleAt(simTime()+timeETP,
    timeOutMsg);
```

Listing 12. SMCD mutant.

```
dataSize = 512000;
mpi_send(getMyMaster(myRank),
    dataSize);◀
```

Listing 13. Original code.

3.2.1. OMCR - OMNET++ method call replacement

The *OMCR* operator replaces a call to the *cancelAndDelete* method by a call to *cancelEvent* or vice versa. Next example presents a mutant obtained by the application of the *OMCR* operator to the code that appears in Listing 15.

3.2.2. SMCR - SIMCAN method call replacement

The *SMCR* operator exchanges *simcan_request_open, simcan_request_close, simcan_request_create* and *simcan_request_delete* method calls. It also interchanges *simcan_request_read* and *simcan_request_write* or *simcan_request_alloc Memory* and *simcan_request_freeMemory*. Listing 18 shows a mutant obtained by the application of the *SMCR* operator to the *simcan_request_write* call in Listing 17.

3.2.3. MMCR - MPI method call replacement

In this case the *MMCR* operator exchanges *mpi_gather, mpi_scatter and mpi_bcast* calls. It also replaces *mpi_send* by *mpi_recv* calls and vice versa. The following example presents a mutant obtained by the application of the *MMCR* operator to the *mpi_send* in the code in Listing 19.

3.3. Shifting operators

The mutation operators included in this category, *OOMU, SOMU, MOMU, OOMD, SOMD* and *MOMD*, shift sentences up or down. Only the calls to methods included in the OMNET++, SIMCAN and MPI APIs can be moved in the model. Listing 22 shows the application of the *OOMD* operator to the call *removeLastModuleFromTrace* that appears in the original code. Listing 24 presents the mutant obtained by the application of the operator *SOMU* to the call *simcan_request_close(path)*; it has been included in the *if* sentence.

3.4. Parameters shuffling operators

The operators *SOSP* and *MOSP* involve swapping actual parameters in methods calls. These changes can be applied in those functions in which the types of the parameters are equal. Taking into account this condition, these operators only can mutate functions in the SIMCAN and MPI APIs. Listing 26 shows the interchange of the actual

```
dataSize = 512000;
◀
```

Listing 14. MMCD mutant.

```
if (!strcmp (msg->getName(),
    SM_WAIT_TO_EXECUTE.c_str()
    )){
      cancelAndDelete(msg);◀
      startState = simTime();
      executeNextState();
  }
```

Listing 15. Original code.

```
if (!strcmp(msg->getName(),
    SM_WAIT_TO_EXECUTE.c_str())
    ){
      cancelEvent(msg);◀
      startState = simTime();
      executeNextState();
  }
```

Listing 16. OMCR mutant.

```
simcan_request_write(outName,
    offWrite, slice_KB*KB); ◀
```

Listing 17. Original code.

```
simcan_request_read(outName,
    offWrite, slice_KB*KB); ◀
```

Listing 18. SMCR mutant.

```
mpi_send(RANK, imageSizeMB);◀
continueExecution();
```

Listing 19. Original code.

```
mpi_recv(RANK, imageSizeMB);◀
continueExecution();
```

Listing 20. MMCR mutant.

```
sm->removeLastModuleFromTrace();
    ◀
send(sm, gateId);
if(updateSentMessages()>=MAX){
    endSimulation();
}
```

Listing 21. Original code.

```
send(sm, gateId);
sm->removeLastModuleFromTrace();
    ◀
if(updateSentMessages()>=MAX){
    endSimulation();
}
```

Listing 22. OOMD mutant.

```
fd=simcan_request_create(path);

if(fd==simcan_OK){
    simcan_request_read(path);
}
simcan_request_close(path);◀
```

Listing 23. Original code.

```
fd=simcan_request_create(path);

if(fd==simcan_OK){
    simcan_request_read(path);
    simcan_request_close(path);
}
```

Listing 24. SOMU Mutant.

```
mpi_send(i, dataSize);
```

Listing 25. original code.

```
mpi_send(dataSize, i);
```

Listing 26. MOSP Mutant.

parameters in a call to the method *mpi_send*. Both parameters have associated the same data type, *integer*.

3.5. Signal swapping operator

This operator, called *OOSS* operator, focuses on swapping the signals of different *cMessage* objects and, therefore, it can only be used used in the OMNeT++ API. In order to apply it, the source code must contain, at least, two *cMessage* creation calls. The following example shows how a mutant (see Listing 28) swaps the signals of two different messages when these are created.

3.6. Parameter replacement operators

The mutation operators of this category replace one parameter in a method call. This type of operator can be applied in the methods provided by the OMNeT++, SIMCAN and MPI APIs. We have designed specific operators for the different APIs and methods to which they can be applied.

3.6.1. OOSNR - OMNET++ Operator signal name replacement

This operator replaces the signal name when a *cMessage* object is created. Hence, this operator can only be applied to the *cMessage* constructor, which is provided by the OMNeT++ API. In the next example, the operator replaces, in the generated mutant (see Listing 30), the signal name by the same name by appending the suffix '@&!' .

3.6.2. OOSTR - OMNET++ Operator scheduling time replacement

This operator replaces the scheduling time of a message to be processed. In this case, this operator can only be applied in the *scheduleAt* method provided by the OMNeT++ API. Next, we provide an example where the mutation operator replaces, in the generated mutant (see Listing 32), the scheduling time by a new one, which is decreased by 30s.

3.6.3. SOFNR - SIMCAN operator file name replacement

This operator replaces the name of a file involved in an I/O operation. Hence, this operator can be applied in those methods related to I/O provided by the SIMCAN API, that is, *simcan_request_read, simcan_request_write, simcan_request_open, simcan_request_close, simcan_request_create* and *simcan_request_delete*. The following example shows a generated mutant (see Listing 34) that performs a read operation in which the file name has been replaced by the same name by appending the suffix '@&!'.

```
cMessage *c;
cMessage *e;
...

c = new cMessage (MCO.c_str());
scheduleAt(simTime()+cDelay,c);

e = new cMessage (MEX.s_str());
scheduleAt(simTime()+eDelay,e);
```

Listing 27. Original code.

```
cMessage *c;
cMessage *e;
...

c = new cMessage (MEX.c_str());
scheduleAt (simTime()+cDelay,c);

e = new cMessage (MCO.s_str());
scheduleAt(simTime()+eDelay,e);
```

Listing 28. OOSS mutant.

```
cMessage *msg;

msg = new cMessage (SIG.c_str())
    ;
```

Listing 29. Original code.

```
cMessage *msg;

msg = new cMessage (SIG.c_str()
    + '@&!');
```

Listing 30. OOSNR mutant.

```
scheduleAt (simTime() +
        connectionDelay,
        waitToConnectMsg);
```

Listing 31. Original code.

```
scheduleAt (simTime() +
        connectionDelay - 30,
        waitToConnectMsg);
```

Listing 32. OOSTR mutant.

3.6.4. SOMAR - SIMCAN operator memory allocation replacement

This operator focuses on replacing the amount of dynamic memory to be allocated. In particular, this operator can be applied in two different method calls, *simcan_request_allocMemory* and *simcan_request_freeMemory*, which are provided by the SIMCAN API. The following example shows a mutant (see Listing 36) where the amount of requested dynamic memory is replaced by 0.

3.6.5. SOCIR - SIMCAN operator communication ID replacement

This operator replaces the actual communication ID in those method calls related to commutations, which are provided by the SIMCAN API. In particular, this operator can be applied to *simcan_request_createConnection, simcan_request_sendDataToNetwork* and *simcan_request_*

```
SIMCAN_request_read (
        fName.c_str(),
        readOffset,
        inputDataSize);
```

Listing 33. Original code.

```
SIMCAN_request_read (
        fName.c_str() + '@&!',
        readOffset,
        inputDataSize);◀
```

Listing 34. SOFNR mutant.

```
SIMCAN_request_allocMemory (
        buffSize,
        GBL_MEM); ◀
```

Listing 35. Original code.

```
SIMCAN_request_allocMemory (
        0,
        GBL_MEM); ◀
```

Listing 36. SOMAR Mutant.

```
simcan_request_sendDataToNetwork
      (sm, dstCommID);◀
```

Listing 37. Original code.

```
simcan_request_sendDataToNetwork
      (sm, dstCommID+5);◀
```

Listing 38. SOCIR mutant.

receiveDataFromNetwork. The next example shows a mutant (see Listing 38) in which the communication ID for sending data to a remote process has been increased by 5.

3.6.6. MOPIR - MPI operator process ID replacement

This operator focuses on replacing the process ID parameter in several method calls provided by the MPI API. In particular, this operator can be applied to *mpi_send, mpi_recv, mpi_bcast, mpi_scatter* and *mpi_gather*. The following example shows a generated mutant (see Listing 40) that replaces the destination process ID in a *mpi_send* call by a new one that is increased by 100.

3.6.7. MOBLR - MPI operator buffer length replacement

Similarly to the previous operator, this one replaces the buffer size parameter in *mpi_send, mpi_recv, mpi_bcast, mpi_scatter* and *mpi_gather* calls, which are provided by the MPI API. The following example shows a generated mutant (see Listing 42) that replaces the buffer size in a *mpi_recv* call by a new one that is decreased by 100.

4. MuTomVo

Testing distributed systems may be an arduous and complex task. There exist several factors that hamper the testing process, among others, the execution of the system against the test cases and the exclusive access to this kind of systems. However, when the usage of these systems is not exclusive, the execution of applications launched by other users also may affect the testing process, causing unfavorable situations such as bottlenecks or delays in the communication network. In order to overcome these problems, we have developed MuTomVo to carry out the testing process in simulated environments. With this goal, we propose a flexible and adaptable framework, where any application that includes API calls or external libraries can be used.

```
mpi_send(procID, buffSize);◀
```

Listing 39. Original code.

```
mpi_send(procID+100, buffSize);◀
```

Listing 40. MOPIR mutant.

```
mpi_recv(procID, buffSize);◀
```

Listing 41. Original code.

```
mpi_send(procID, buffSize-100);◀
```

Listing 42. MOBLR mutant.

MuTomVo is a mutation testing framework which provides mechanisms to generate test suites and evaluate their effectiveness to check distributed systems. MuTomVo allows to apply the mutation operators previously proposed for reproducing the common mistakes made by competent programmers developing distributed applications.

The main components of MuTomVo, such as the architectural design, the mutation engine, the testing process and the methods for automatic tests generation, are described in this section.

4.1. Architectural design

The architecture of MuTomVo is illustrated in Fig. 2. This scheme depicts how the mutation testing techniques are integrated with modeling and simulation tools. Also, we provide a detailed description of the required steps to perform the testing process, whose goal is to estimate the effectiveness of a test suite to detect errors in an application running in a simulated distributed environment.

Initially, the user must build a system model using the *GUI* (*Graphical User Interface*) SIMCAN simulator (see Fig. 1). This model is composed of two main components. The first one, known as *simulation scenario*, is the configuration of the distributed architecture, including physical machines and network connections, which are used by the simulation platform to deploy the system. The second one is the application that will be executed over the provided architecture. In order to facilitate this process, SIMCAN offers a repository of predefined modules and a set of applications which can be used by the user to compose its own model ①. These modules simulate the behavior of the elements, such as CPUs, disks, memories and communication networks that comprise the distributed system. The applications that are included in the repository cover different paradigms of data processing and communications, such as Map-Reduce, client-server and single node computation. In addition, new applications can be modeled by the user using the APIs provided by the simulation platform. Once the design of the model is ended, MuTomVo generates the configuration files required by SIMCAN to simulate the designed environment ②. Next step ③ consists in providing a test suite for testing the application. The test cases can be either created manually by the user or generated automatically by MuTomVo by means of two different techniques. Following, the mutation engine must be configured ④. The user must establish the required information to perform the mutation process. This information corresponds to the application designed in ②, the test suite built in ③ and the mutation operators that will be applied to the application for generating the mutants. Once the configuration phase is completed, the mutation engine receives the application and starts the mutant generation process ⑤. At this point, the mutation engine applies the selected mutation operators to inject different faults in the original code of the provided application for creating faulty versions, that is, mutants ⑥. At this point, the framework has the system model, the application and the generated mutants. SIMCAN uses the system model ⑦ to build the architecture and topology configured at step ①. Next, all the mutants are compiled and both the original application and the mutants are executed against the test suite in SIMCAN. The

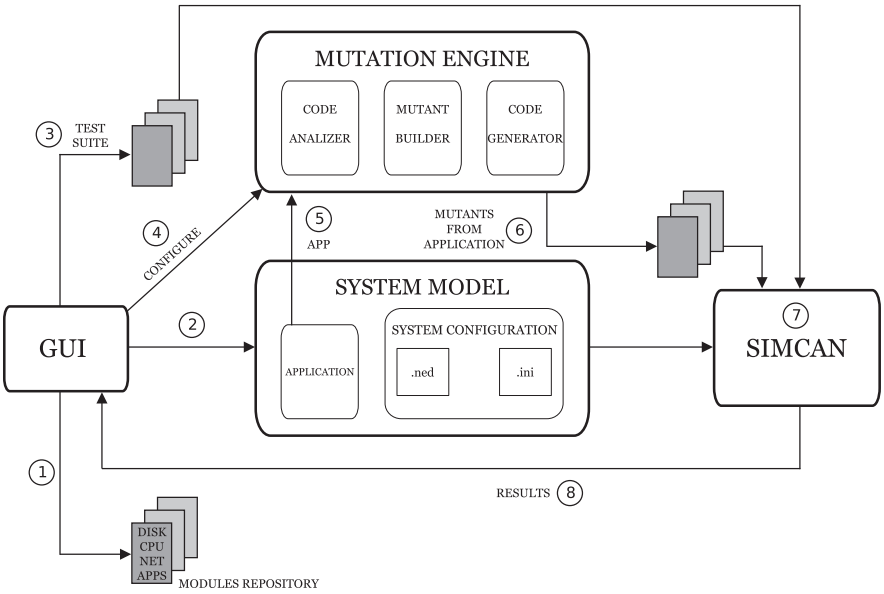


Fig. 2. Architecture of MuTomVo.

results obtained from the execution of the original application are compared with the ones obtained from each mutant execution in order to check if the mutants are killed or alive. Finally, the results are sent to the GUI module to present them to the user ⑧.

4.2. Mutation engine

The continuous development of new contributions in the mutation testing field, requires to have a flexible environment for being extended with new operators and optimization techniques. Therefore, MuTomVo is built on a modular and flexible architecture.

On the one hand, MuTomVo is modular in the sense that its functionality is divided into independent modules. Fig. 3 shows the four modules used to carry out mutation analysis: *mutation engine, code analyzer, mutant builder* and *code generator*. Consequently, different modifications can be applied to each module without interfering with the rest of the framework. On the other hand, our proposed framework is flexible in the sense that different approaches can be integrated in each module. For instance, different code analyzers can be used to process the source code of different programming languages like, among others, C, C++ and Java. These features easily allow the inclusion of both existing and new techniques, such as selective mutation (Delgado-Pérez et al., 2017), trivial compiler equivalence (Papadakis et al., 2015) and mutation clustering (Ma and Kim, 2016). This aspect reduces the integration time significantly, and increases the feasibility of having a high number of techniques in the framework.

The main element, *mutation engine*, is the responsible for orchestrating the communication among the other components of the framework. This module provides a high level of flexibility, allowing to easily include new mutation operators. When the mutation process starts, the *mutation engine* receives as input the application to be mutated and transfers it to the *code analyzer* ①. The *code analyzer* module analyzes the source code in order to determine possible mutation operators targets. With this goal, the Eclipse C/C++ development tooling parser (CDT) (Piatov et al., 2012) has been integrated in the tool. CDT parser provides mechanisms to facilitate the code analysis, such as abstract syntax tree that saves the user building its own syntactical analyzer. In addition, CDT has an open source license and is actively maintained by the Eclipse community. All of that makes CDT a robust and major asset for the active and flexible development of MuTomVo. The data structures generated by the code analyzer are provided to the *mutant builder* ②, that generates the mutants and saves them in memory. Let us remark that the mutant builder does not generate syntactically invalid mutants. It takes into account several factors depending on the mutation operator applied, such as the data type of the parameters of the methods calls affected by the *shuffle parameters* or the *replace* operators. Finally, the *code generator* creates and stores in disk ③ the source code obtained in the previous step. This module is based on a layered architecture where the first layer implements basic operations, such as export the source code to disk or mutant enumeration, and the second layer extends it by using specific source code generators: a standard code generation and an API-based code generation.

4.3. Tests cases

In this framework, a test case corresponds to a set of pairs $< parameter, value >$ that configures the execution of the application and its mutants, such as the computing amount (in MIs) and the size of processed data (in GB). The structure of the test cases for each specific application is provided by the user. It must contain all the necessary parameters for the simulation of the application. All the test cases will associate a value to each of the parameters.

For example, let us consider the application *appCpu* used in Section 5.2.1. It performs different operations over a data set. Basically, while the data set is not processed entirety, the application reads a piece of data, performs a computation and writes the result to a file. The Listing 43 shows a test case for this application. All the required parameters for simulating the execution of the application and the mutants have assigned a value. In this case the parameters denote the size of the data set to be processed (*inputDataSize*), the size of the file that stores the results (*outputDataSize*), the computation of a piece of data in millions of instructions (*MIs*) and the number of times the

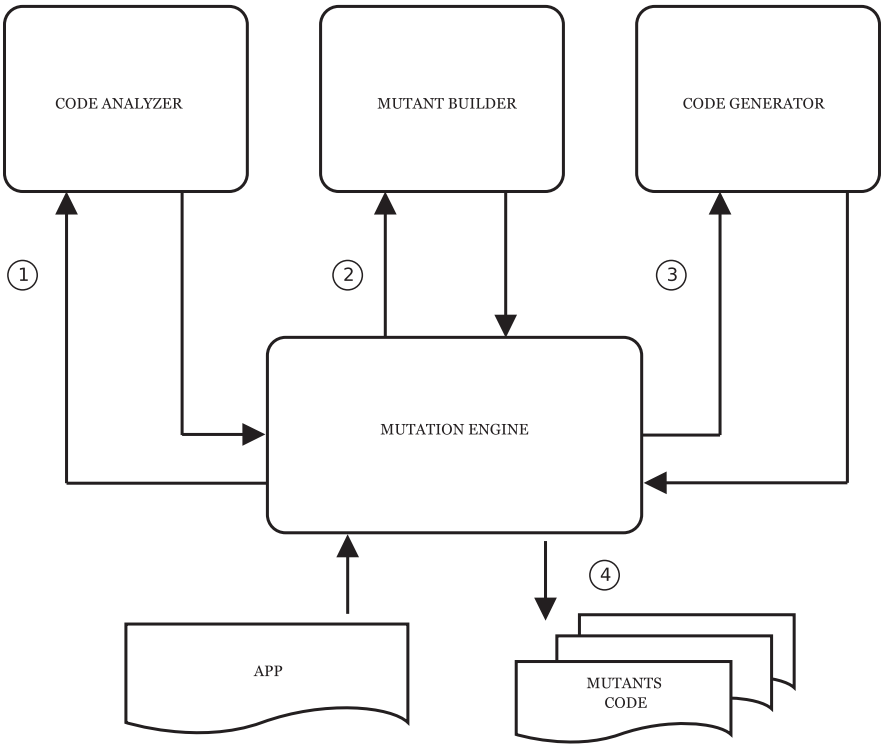


Fig. 3. Mutation engine architecture.

application will be executed (*iterations*).

In addition to the structure of the test cases, that only consists of the input parameters, the tester also has to provide the system with the output parameters that must be returned by the simulation of the application and the mutants. The values returned by the execution of the original application against the test cases will be used as *oracle*. If the values returned by the execution of the mutant are not equal to the ones produced by the original application, the mutant is killed, in other case, it is alive.

In our example, the output parameters correspond to the time spent in input/output operations during the execution (*ioTime*), the CPU time (*cpuTime*) and the simulation time (*simTime*). Listings 44 and 45 show the values returned by the execution of the application and a mutant against the previous test case, respectively. In this case, the difference between the values indicates that the mutant has been killed by the test case.

If either the values of the output parameter returned by a mutant does not correspond to the ones produced by the original application or the execution time exceeds the established timeout, the mutant is not executed against the rest of the test cases and it is classified as killed.

The generation of test cases can be done manually by using the MuTomVo GUI. However, providing a test suite that comprehensively checks an application is a very expensive and error-prone task. In order to overcome these difficulties, we have developed a method for automatically generate test suites.

This method consists in generating a random test suite. Random testing is a well-known testing technique that, despite its simplicity, is widely used in both real-world and scientific applications (Arcuri et al., 2012; Ciupa et al., 2011). The configuration parameters included in the structure of the test cases is used to create a collection of instances. In addition, the user must select the set of parameters that will be randomly valued, the maximum and minimum values that can take each parameter and the total number of tests to be generated.

```
ioTime=14.857143;
cpuTime=0.720575;
simTime=15.577717;
```

Listing 44. Original program output.

```
ioTime=10.785714; ◀
cpuTime=0.330697; ◀
simTime=11.116411;◀
```

Listing 45. Mutant output.

```
int inputDataSize=19MiB;          // Size of data-set
int outputDataSize=8MiB;          // Size of results
int MIs=698667;                   // Computing for each iteration
int iterations=5;                 // Number of iterations
```

Listing 43. Example of test case.

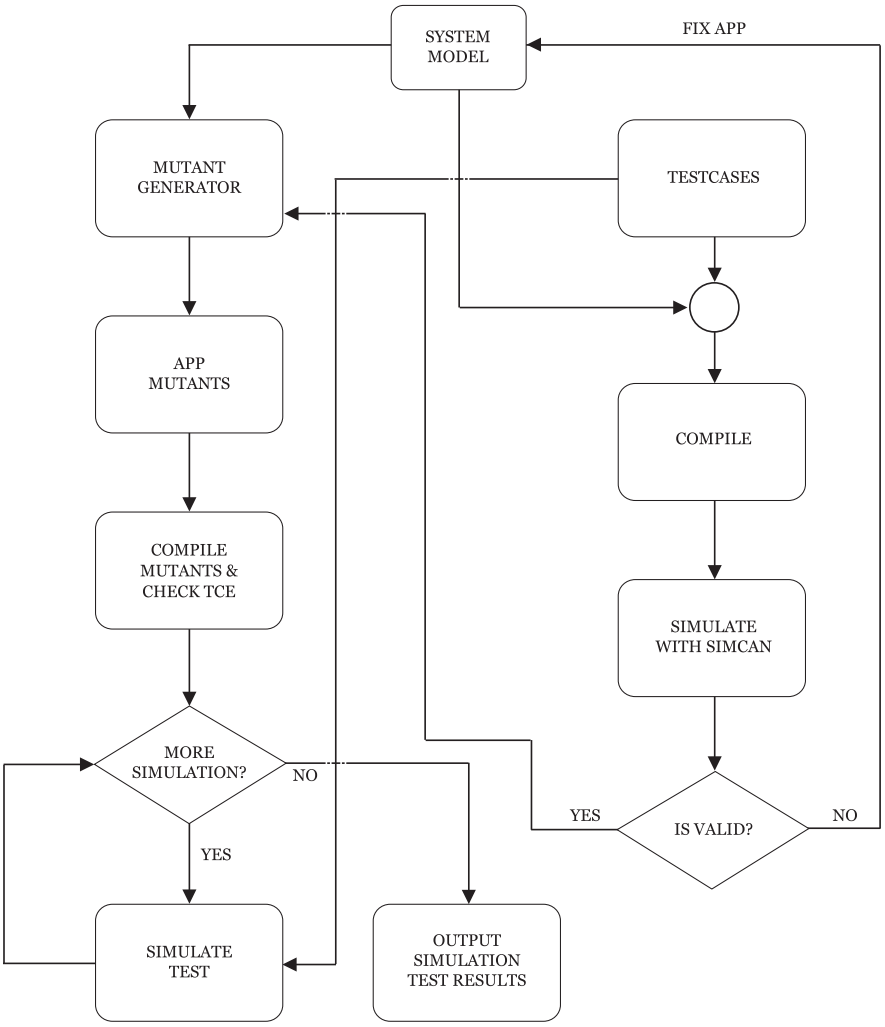


Fig. 4. Testing process.

4.4. Testing process

Fig. 4 illustrates the mutation generation and execution processes. Initially, the process starts when the user provides MuTomVo with three elements: a system architecture model, an application to be executed in this architecture and a test suite for checking the correctness of the application. The first task of the testing process is the compilation of the application, carried out by the GCC compiler.

In order to automatically identify both *equivalent* and *duplicated* mutants, the TCE (Trivial Compiler Equivalence) approach has been included in the testing process (Papadakis et al., 2015). A mutant is considered *equivalent* when it is semantically identical to the original program and there is no test case that can kill it. Similarly, *duplicated mutants* are considered a special form of equivalent mutants in a way that they are equivalent each another, but not to the original program. Basically, TCE detects equivalent mutants by compiling each mutant and comparing its executable file with the executable file of the original program. We assume that a mutant is equivalent when both executable files are identical. Similarly, TCE also detects duplicated mutants by comparing executable files generated from different mutants. Hence, both equivalent and duplicated mutants are removed from the generated mutant set.

Once the compilation phase has finished, the execution of the original application against each test case is simulated in SIMCAN. In order to ensure the validity of the provided system model, the produced outputs are checked; in case that the output reveals a configuration error, both the application and the architecture must be revised to fix the system model. There are different types of errors that can be identified in a simulated environment, those are, among others:

- Architecture misconfiguration. The topology of the distributed system is not well configured, being some of the components

disconnected from the communication network.
- The application does not finish its execution. In this case, the user must manually set a timeout to stop the simulation. Since it is undecidable (see The Halting Problem) the user must know the application under test and then set a pre-defined timeout depending on the values used in the test case.
- The application accesses a file that does not exist in the file system of the simulated environment.
- The application does not allocate enough memory. Then, when the user writes some information in the memory an exception is thrown.
- Appearance of unhandled exceptions. The application throws unmanaged exceptions, like division by zero, that are not captured by the user.

On the contrary, if none of the errors described above occur and the outputs are consistent, the tool will proceed with the generation of the mutants. Let us remark that MuTomVo has been designed to generate valid mutants in the sense of producing syntactically correct programs to be successfully compiled. After the mutant compilation all of them are executed sequentially against each test case. If either the values of the output parameter returned by a mutant does not correspond to the ones produced by the original application or the execution time exceeds the established timeout, the mutant is not executed against the rest of the test cases and it is classified as *killed*. Otherwise, if the all the outputs produced by the execution of a mutant against all the test cases are equal to the ones produced by the application, it is said that the mutant is *alive*. At the end of this process the results are shown to the user through the MuTomVo GUI.

5. Empirical study

This section describes the evaluation process carried out to determine the applicability of the proposed framework. The main goals of these experiments are to analyze the effectiveness of test suites for detecting errors in distributed applications and measure the usefulness of the proposed mutation operators.

5.1. Research questions

The experiments described in this section seek to answer the following questions:

- **RQ1**: How effective are the proposed mutation operators for generating high-quality mutants?
  The main principle of mutation testing consists on injecting faults, which represent common mistakes that programmers often make, in the application under test. Hence, we say that a mutation operator is effective if it generates real errors in the application under test and, therefore, it minimizes the number of generated equivalent mutants. We are interested in comparing the number of equivalent mutants obtained when the testing process applies the set of mutation operators that we propose with the number of equivalent mutants generated when the traditional operators are applied.
- **RQ2**: How appropriate is the mutation score for a test suite applied to a mutant set generated by using our proposed mutation operators?
  In mutation testing, the quality of a test suite is measured by a criterion called *mutation score*. In essence, the *mutation score* calculates the effectiveness of a test suite in terms of its ability to detect faults. We investigate whether the same test suite provides different results when it is applied against different mutant sets, each of them generated by using different mutation operators. Also, we are interested in analyzing the correlation between the quality of the generated mutants and the obtained mutation score.
- **RQ3**: How scalable is our approach to perform the testing process in distributed applications?
  The selected applications to carry out the experiments of this paper cover three well-known paradigms, that is, client/server paradigm, scientific applications that mix CPU-intensive and data-intensive paradigms, and the pipeline paradigm. It helps us to determine the effectiveness and efficiency achieved in the testing process using different distributed applications.

5.2. Experimental design and procedure

The selected applications to carry out the experiments cover three well-known paradigms used in both cloud and HPC infrastructures, these are, client/server paradigm, scientific applications that mix CPU-intensive and data-intensive paradigms, and the pipeline paradigm (Schad, 2010; Jackson et al., 2010; Li et al., 2010).

These applications have been developed and deployed in three different simulated data centers. For each of them we have generated both a set of mutants as result of applying the proposed mutation operators and a specific test suite to check the application. The generation of the test suites has been performed by using the random method described in Section 4.3. All the test suites contain 100 test cases. The application of the mutation operators to the applications has produced 514 mutants that have been executed against the tests cases, that means a total of 51,400 simulations. This engine uses an abstract syntax tree that aids in the injection of errors in such a way that only syntactically correct mutants are generated.

The effectiveness of a test suite is measured on the basis of the *mutation score* (MS), that is, the percentage of non-equivalent mutants that the test cases have killed. We have classified the results by the mutation operator that generated the mutants and the total time required to execute each test case. The latter criterion will help us to determine the test cases that kill the maximum number of mutants but spend the minimum time in the testing process.

In our experiments 102 equivalent mutants have been found. Initially, we use TCE to automatically detect equivalent mutants. However, this technique was not able to identify all of them and we had to do it manually. The time invested to analyze and identify each of them did not take more than 3 minutes. The highest number of equivalent mutants was generated by the OMNeT++ operators due to their reduced impact on the behavior of the applications. The MuTomVo tool, which implements the proposed framework, has been used to automatically apply the mutation testing process. In order to alleviate the high computational cost of the testing process, the EMINENT algorithm (Cañizares et al., 2016) implementing the OUTRIDER optimization (Cañizares et al., 2017) has been used.

All the experiments were performed on a 8-node cluster, where each node is provided by a Quad-Core Intel(R) Core(R) i5-3470 CPU at 3.4 Ghz with hyper-threading, 8 GB of RAM and 500 GB HDD. These nodes are interconnected through a Ethernet Gigabit network.

5.2.1. Client/server application

The first application, known as *appCPU*, consists of 300 lines of code and performs massive computational operations over a data set. Initially, the data set is stored in a remote server. Then, the application loads this data set in local memory to perform the corresponding operations. Once the data set has been processed entirely, the results are stored in the remote server. Fig. 5 depicts the distributed system where *appCPU* has been deployed.

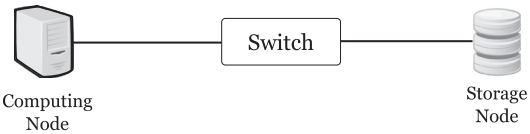


Fig. 5. Diagram of the *appCPU* application.

```
int inputDataSize;          // Size of data-set
int outputDataSize;         // Size of results
int MIs;                    // Computing for each iteration
int iterations;             // Number of iterations
```

Listing 46. Configuration parameters of the *appCPU* application.

```
string testID;          // Identifier of the test executed
double ioTime;          // Total time spent in i/o
double cpuTime;         // Total time spent in cpu
double simTime;         // Total simulation time
```

Listing 47. Output parameters of the *appCPU* application.

The simulated environment in which the *appCPU* application has been executed consists of two computers (one of them corresponds to the client and the other one to the server) with Dual-Core Intel(R) Xeon (R) CPU, 4 GB of RAM, a Gigabit Ethernet interface, and a 500 GB hard disk.

The parameters required to configure the application are detailed in Listing 46. As we introduced in Section 4, the parameter *inputDataSize* indicates the total size of the data set, *outputDataSize* denotes the total size of the results file, *MIs* represents the computation measured in millions of instructions and *iterations* refers to the number of data sets processed by the application. Listing 47 shows the output parameters required to check if a mutant has been killed or not by a test case. The parameter *testID* represents the identifier of the test applied in the execution of the application/mutant, *ioTime* denotes the time spent in input/output operations, *cpuTime* contains the cpu time and *simTime* corresponds to the simulation time. The killing condition for this application focuses in three values, that is, time invested in CPU operations (cpuTime), time invested in I/O operations (ioTime) and the total execution time (simTime). The mutant is killed if, at least, one of these values obtained from its execution differs from the values obtained in the execution of the original application.

Although we applied all the proposed mutation operators during this experiment, due to the simplicity of the source code, only 11 of them, concerning OMNeT++ (*OOMU, OMCD, OOSNM, OOSTM and OOPD*) and SIMCAN (*SOMU, SOMD, SOSP, SMCD, SMCR and SOFNR*) method calls, have generated a total of 27 mutants.

Table 3 registers the mutation scores obtained in this experiment. Out of a total 27 of mutants, 22 (81.48%) were killed and 5 (18.52%) were identified as equivalent. In this case TCE have not detected equivalent mutants and, therefore, these were manually identified. Specifically, the equivalent mutants were generated by the *OOSTM, SOMU* and *SOMD* operators.

Fig. 6 shows a *killmap* that provides information of the state of each mutant (killed or alive) after its execution over each test case. The *x*-axis represents the test ID and the *y*-axis shows the mutant ID. The

Table 3
Results of *appCPU* application.

Operator	Mutants				MS(%)
	generated	Alive	Killed	Equivalent	
OMCD	1	0	1	0	100
OOPD	6	0	6	0	100
OOMU	1	0	1	0	100
OOSNR	1	0	1	0	100
OOSTR	3	0	1	2	100
OMNeT++ operators	**12**	**0**	**10**	**2**	**100**
SMCD	3	0	3	0	100
SMCR	3	0	3	0	100
SOMU	3	0	2	1	100
SOMD	2	0	0	2	100
SOSP	2	0	2	0	100
SOFNR	2	0	2	0	100
SIMCAN operators	**15**	**0**	**12**	**3**	**100**
Total	**27**	**0**	**22**	**5**	**100**

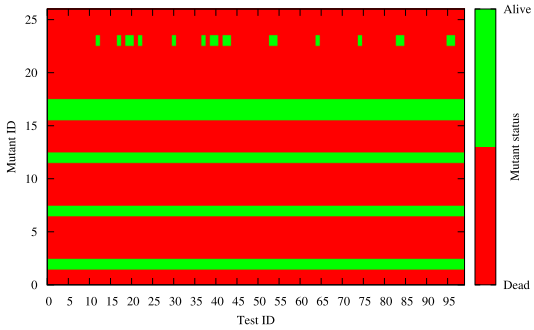


Fig. 6. Killmap of the *appCPU* application.

```
// Reset timer!
startServiceCPU = simTime ();
//Call SIMCAN API
SIMCAN_request_cpu(MIs);
```

Listing 48. Original Code.

```
// Reset timer!
SIMCAN_request_cpu(MIs); ◄
//Call SIMCAN API
startServiceCPU = simTime ();
```

Listing 49. SOMU Mutant.

results show that 22 mutants out of 27 were killed. Listing 49 shows an example of an equivalent mutant. Listing 48 shows the portion of code from the application under test where the operator *SOMU* is applied. In this case, both applications are equivalents due to the fact that the swap of the sentence that performs a CPU call and the sentence that only carries out a reset a timer, does not has direct impact in the result of the execution.

5.2.2. Intensive computation application

The second application, known as *appMR* (Núñez and Merayo, 2014), is a simplified version of the Map-Reduce model proposed by Google (Dean and Ghemawat, 2008), which consists of 900 lines of code. The main goal of this application is to process, in parallel, a data set by distributing pieces of data among multiple processes. The application uses two types of processes, coordinators and workers. These processes are grouped in frames, in such a way that each frame contains a coordinator process and several worker processes. Fig. 7 illustrates the basic scheme of this application. The execution starts by reading a piece of the data set, denoted as domain, from the storage nodes ①. This action is carried out by the coordinator processes. Next, each coordinator process sends a slice of the domain to each worker process ②. Each worker processes the received domain ③ and the results are sent to the corresponding coordinator process ④. Finally, the

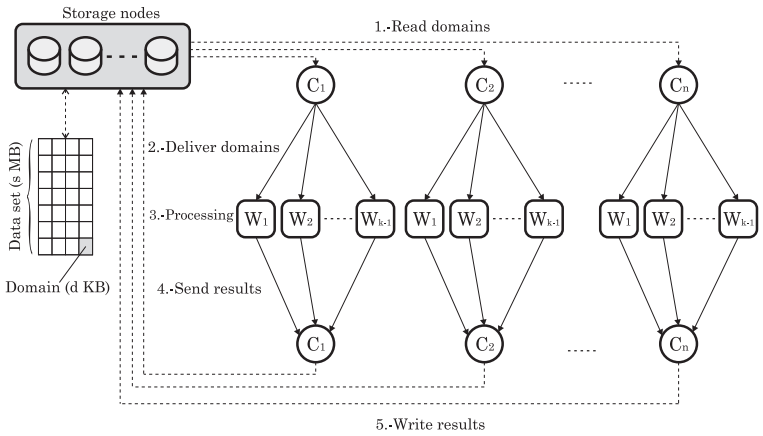


Fig. 7. Diagram of the *appMR* application.

coordinator stores the results ⑤ and a new domain is read ①. The execution finishes when the data set is processed entirely.

The simulated environment in which the *appMR* application has been executed is composed of 64 computational nodes and 8 storage nodes. Each node is powered by a Dual-Core Intel(R) Xeon(R) CPU and 4 GB of RAM, a Gigabit Ethernet interface, and a 500 GB hard disk. The topology of the environment is a Top Of Rack network.

The parameters required to configure this application are shown in Listing 50. The *workersSet* denotes the number of processes for each frame, *sliceToWorkers* and *sliceToMaster* correspond to the size of each domain (measured in KBytes) that is sent and received, respectively, by the workers and the master, *sliceCPU* provides the number of computational operations performed by the workers to process a slice of data measured in MIs, *workersWrite* indicates if the worker processes can write the results to disk instead of sending them to their coordinator processes, *workersRead* indicates if the worker processes can read the data set from storage and, finally, *numIterations* represents the number of iterations. Listing 51 shows the output parameters required to determine if a mutant has been killed or not by a test case. The parameter *testID* represents the identifier of the test applied for the execution of the mutant and *procID* corresponds to the identifier of the process. The latter parameter is necessary due to the fact that in this application there exist several processes running in parallel and all of them must be identified. The rest of output parameters, *ioTime, cpuTime, netTime* and *simTime*, correspond to the time spent in input/output operations during the execution, the cpu time, the time spent in communications and the simulation time, respectively.

The killing condition for this application focuses in four values, that is, time invested in CPU operations (cpuTime), time invested in I/O operations (ioTime), time invested in network operations (netTime) and the total execution time (simTime). Hence, a mutant is kept alive if all the values obtained from its execution are equal to the values produced by the execution of the original application. The mutant is killed otherwise.

In this case, the mutation operators generated 126 mutants. As we can see in Table 4, all the mutation operators produced mutants. The operator *MMCR* generated the highest number of mutants (19.04%) followed by SMCR (8.73%). Out of 126 mutants, 90 (90.47%) were killed and 13 (10.31%) were identified as equivalents. In this case 4 duplicated mutants were automatically detected by aplying TCE, while 9 equivalent mutants were manually detected. It is worth noting that most of the equivalent mutants were generated by OMNeT++ mutation operators. This is due to the low impact that these operators had in the behavior of this application. Regarding SIMCAN operators, 2 equivalent mutants were generated by the *SOMU* operator. The only mutation operator, focusing on MPI calls, that produced an equivalent mutant is *MOMD*. All the non-equivalent mutants were killed by the test suite and, therefore, all the operators show a mutation score of 100%.

Fig. 8 depicts the *killmap* of this experiment. On the contrary to the *appCPU* application, in this case, not all the test cases are able to kill all the non-equivalent mutants. There exist several mutants that are killed only by a subset of the test suite. The alive mutants were identified as equivalent.

Listing 53 shows an example of an equivalent mutant. Listing 52 shows the portion of code from the application under test where the operator *MOMD* is applied. In this case, both applications are considered equivalents because the swap of these sentences, that sends data over the communications network (*mpisend*(*getMyMasterID*(*processID*), *dataSize*)) and calculates the next state of the state machine (*calculateNextState*()), does not produce a different result.

5.2.3. Pipeline application

The last application, called *appPIPE*, is a pipeline image processing which deals with a set of X-Ray images of human skulls to automatically detect the frontal sinus (Merayo and Núñez, 2015). It consists of 3500 lines of code. Basically, the application divides a set of processes among the nodes of a distributed system. Each process performs a specific task: a process manages the images stored in the database node, several processes are in charge of analysing the images for pre-processing, filtering and detecting bottom and top borders of the frontal sinus and the rest of the processes use a majority voting phase with the goal of giving

```
int workersSet;          // Number of worker processes per frame
double sliceToWorkers;   // Slice of data sent to each worker process
double sliceToMaster;    // Slice of data received for master process
int sliceCPU;            // CPU processing for each worker process
bool workersWrite;       // Worker processes write results on disk
bool workersRead;        // Worker processes read results on disk
int numIterations;       // Number of iterations
```

Listing 50. Configuration parameters of the *appMR* application.

```
string testID;          // Identifier of the test executed
double procID;          // Process identifier
double ioTime;          // Total time spent in i/o
double cpuTime;         // Total time spent in cpu
double netTime;         // Total time spent in network
double simTime;         // Total simulation time
```

Listing 51. Output parameters of the *appMR* application.

Table 4
Results of *appMR* application.

Operator	Mutants				MS(%)
	generated	Alive	Killed	Equivalent	
OMCD	2	0	0	2	–
OOPD	7	0	6	1	100
OMCR	2	0	0	2	–
OOMU	2	0	1	1	100
OOMD	2	0	0	2	–
OOSS	1	0	1	0	100
OOSNR	2	0	2	0	100
OOSTR	6	0	4	2	100
OMNeT++ operators	**24**	**0**	**14**	**10**	**100**
SMCD	8	0	8	0	100
SMCR	11	0	11	0	100
SOMU	7	0	5	2	100
SOMD	6	0	6	0	100
SOSP	4	0	4	0	100
SOFNR	6	0	6	0	100
SOMAR	1	0	1	0	100
SIMCAN operators	**43**	**0**	**41**	**2**	**100**
MMCD	6	0	6	0	100
MMCR	24	0	24	0	100
MOMU	6	0	6	0	100
MOMD	5	0	4	1	100
MOSP	6	0	6	0	100
MOPIR	6	0	6	0	100
MOBLR	6	0	6	0	100
MPI operators	**59**	**0**	**58**	**1**	**100**
Total	**126**	**0**	**113**	**13**	**100**

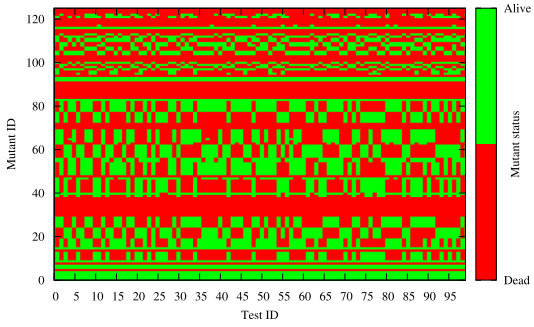


Fig. 8. Killmap of the *appMR* application.

```
// Send data to each process
mpi_send (getMyMasterID(
    processID), dataSize);
//Calculate the next state
calculateNextState ();
```

Listing 52. Original code.

a verdict about the detection of the top or bottom borders.

The simulated environment in which the application has been deployed is composed of one storage node to host a database, computing nodes to execute processes, a communication network to interchange information among nodes and a communication switch. Fig. 9 shows a

```
// Send data to each process
calculateNextState ();
//Calculate the next state
mpi_send (getMyMasterID(
    processID), dataSize); ◀
```

Listing 53. MOMD mutant.

detailed configuration of the system, which is made up of 11 nodes:

- 1 database node: This node stores the database. It includes an AMD Athlon CPU with 2.2 GHz, 4 GB of RAM memory and a RAID 0 system with 4 hard disks of 1 TB.
- 4 main nodes (grey nodes): These nodes include a CPU Intel Xeon with 2 Ghz, 4 GB of RAM memory and a hard disk of 1 TB.
- 6 voting nodes (black nodes): These nodes include an AMD Athlon CPU with 2.2 GHz, 2 GB of RAM memory and a hard disk of 160 GB.
- All nodes are interconnected using a switch through an Ethernet Gigabit network.

The parameters required to configure this application are shown in Listing 54. The parameter *workersSet* indicates the number of processes that take part in the image recognition process, *imageSizeMB* corresponds to the size of each image, *numberOfImages* indicates the number of images stored in the database and *totalPrefetchImages* represents the number of images that the buffer of each node can keep. A second group of parameters are related to the quantity of computational operations used in the different stages of the recognition process, among others, *processingCPU, filteringCPU* and *bottomCPU* that correspond to the computing required, in MIs, for preprocessing, filtering and locating the cranium bottom of each image, respectively. Finally, there is a set of parameters used to establish timeouts for the different stages of the process, like *timeoutWTP, timeoutETP* and *timeoutECT* that give, respectively, the maximum time allowed for filtering an image, receiving a filtered image to detect the bottom border and obtaining the result of the checking process performed before the filtering stage.

The killing condition for this application focuses in its execution trace (Merayo and Núñez, 2015). Each operation is registered in a trace file, which contains a time stamp, the name of the operation and the corresponding parameters. Thus, the execution of the original application produces a trace that is compared with the trace generated by the mutant. The mutant is killed if these traces are not equal.

In the experiment performed with this application were generated 361 mutants. On the contrary to the previous experiments in which all the test cases were obtained automatically by using the random method, in this case, due to the complexity of the application under test, 17 out of 100 were designed by hand. It is well known the benefits obtained with the use of random testing used alongside with manual tests (Ciupa et al., 2011).

As we can see in Table 5, all the mutation operators produced mutants. The operator *OOSTM* generated the highest number of mutants (13.29%) followed by *MMCR* (8.86%) and *OMCD* (7.75%). Out of 361 mutants, 267 (73.96%) were killed and 82 (22.71%) were identified as equivalents. In addition, 13 duplicated mutants were automatically detected by using TCE and 69 mutants had to be manually identified as equivalents. The highest percentages of equivalent mutants were generated by mutation operators *OOMD, OOMU* and *OMCR*, with 95%, 72.72% and 35.71%, respectively. In this experiment, four operators, *OMCD, SMCD, MOPIR* and *MOBLR*, generated mutants that

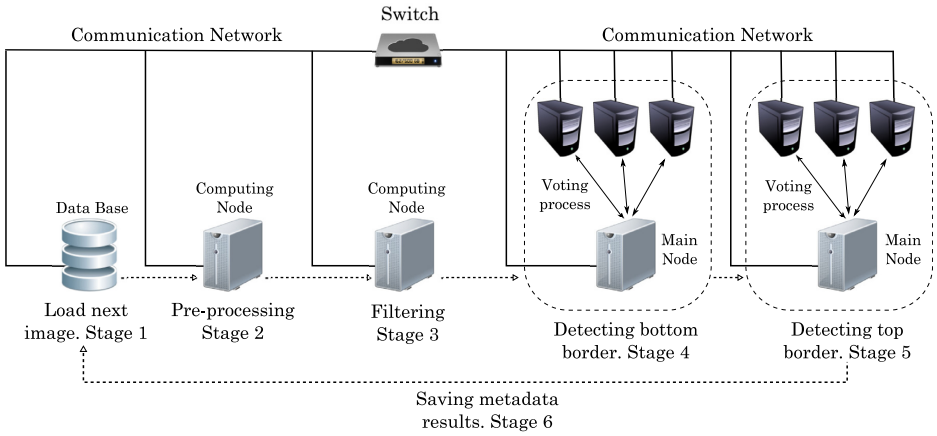


Fig. 9. Stages and architecture of the *appPIPE* application.

the test cases were not able to kill, with mutation scores of 72, 87, 95 and 95, respectively.

The *kilmap* of this experiment it is shown in Fig. 10. The first 17 tests correspond to those created by hand. This chart shows that a test suite consisting of tests generated both by hand and randomly kills a higher number of mutants than a test suite only consisting of randomly generated tests. We must say that most of the alive mutants were generated by the shift-up and shift-down operators. Listing 56 shows an example of an equivalent mutant. Listing 55 shows the portion of code from the application under test where the operator *OMCD* is applied. These applications are considered equivalents because the sentence that cancels and deletes a message (*cancelAndDelete*(*msg*)) does not affect the result of the execution.

It is important to remark that during the experimental study we have detected an error in the pipeline application. In particular, one timeout parameter was misconfigured. Consequently, this parameter was never evaluated in the source code and specific portions of the code were never executed.

5.3. Case study using traditional mutation operators

This section presents an experiment where traditional mutation operators are applied during the testing process for generating mutants. The test suites and applications under test used in this experiment are the same as the ones described in Section 5.2.

Table 6shows an overview of the obtained results. The first column refers to the mutation operator used to generate the mutants. The next four columns refers to the aggregate number of mutants for the three applications. The last column gives the obtained mutation score.

In this experiment a total of 4019 mutants have been generated by using 10 mutation operators. The number of alive and killed mutants is 425 (10.68%) and 1911 (47.54%), respectively. However, it is interesting to highlight the high number of generated equivalent mutants, which in this case reaches 1683 (41.87%). We applied TCE this set of mutants to detect the number of equivalent and duplicated mutants. In this case, TCE automatically detected 1059 duplicated and 95 equivalent mutants. However, in order to provide accurate results, we manually checked those equivalent mutants that have not been detected by TCE.

The obtained mutation score ranges from 54.1, using a mutant set generated by the AORs operator, to 92, using a set of mutants generated by the COD operator. In average, the test suites executed over all the generated mutants provide a mutation score of 81.8%.

Since these results may seem acceptable, the substantial effort and time required for detecting the equivalent mutants makes this option unfeasible.

5.4. Analysis of the results

In this section we discuss the obtained results from the empirical study using the mutation operators described in Tables 1 and 2. For the sake of clarity, we summarize these results in Table 7, which shows the aggregate number of mutants generated by each mutation operator and the mutation score. Table 8 depicts and overview of the obtained results

```
int workersSet;                 // Number of worker processes
int imageSizeMB;                // Total size of the processed image
int numberOfImages;             // Number of images stored in the DDBB
int totalPrefetchImages;        // Buffer size for each node
int processingCPU;              // MIs for processing each image
int filteringCPU;               // MIs for filtering each image
int bottomCPU;                  // MIs for locating the cranium bottom
int topCPU;                     // MIs for locating the cranium top
int votingCPU;                  // MIs for voting
double timeoutWTP;              // Timeout for WTP
double timeoutETP;              // Timeout for ETP
double timeoutECT;              // Timeout for ECT
double timeoutWTF;              // Timeout for WTF
double timeoutETF;              // Timeout for ETF
double timeoutWCT;              // Timeout for WCT
double timeoutVotingBottom;     // Timeout voting the bottom part
double timeoutVotingTop;        // Timeout voting the top part
```

Listing 54. Configuration parameters of the *appPIPE* application.

Table 5
Results of *appPIPE* application.

Operator	Mutants				MS(%)
	generated	Alive	Killed	Equivalent	
OMCD	28	6	16	6	72
OOPD	18	0	17	1	100
OMCR	14	0	9	5	100
OOMU	22	0	6	16	100
OOMD	20	0	1	19	100
OOSS	3	0	3	0	100
OOSNR	2	0	2	0	100
OOSTR	48	0	34	14	100
OMNeT++ operators	**155**	**6**	**88**	**61**	**93.6**
SMCD	15	2	13	0	87
SMCR	11	0	11	0	100
SOMU	15	0	11	4	100
SOMD	6	0	0	6	100
SOSP	3	0	3	0	100
SOFNR	6	0	6	0	100
SOCIR	4	0	4	0	100
SIMCAN operators	**60**	**2**	**48**	**10**	**96**
MMCD	23	0	23	0	100
MMCR	32	0	32	0	100
MOMU	17	0	12	5	100
MOMD	22	0	16	6	100
MOSP	8	0	8	0	100
MOPIR	22	1	20	1	95
MOBLR	22	1	20	1	95
MPI operators	**146**	**2**	**131**	**13**	**98.5**
Total	**361**	**10**	**267**	**84**	**96.3**

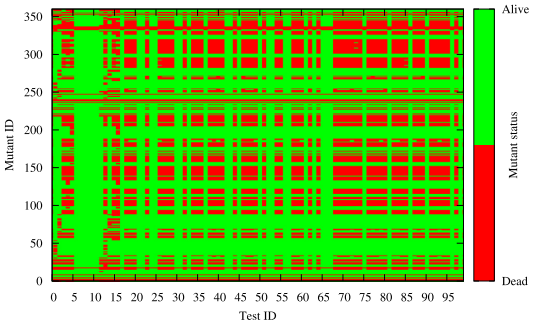


Fig. 10. Killmap of the *appPIPE* application.

```
// Delete message
cancelAndDelete (msg);

// Establish all connections...
establishAllConnections();
```

Listing 55. Original code.

```
// Delete message
;

// Establish all connections...
establishAllConnections();
```

Listing 56. OMCD equivalent mutant.

during the testing process. We also answer the research questions.

In general terms, the results obtained using our proposed framework are promising, achieving an average mutation score of 0.975 (see Table 7). In this study, we have used three applications with different degrees of complexity, ranging from 200 to 3500 lines of code.

Table 6
Results of traditional operators.

Operator	Mutants				MS(%)
	generated	Alive	Killed	Equivalent	
AORb	392	26	234	132	90
AORs	72	33	39	0	54.1
AOIu	796	96	353	347	78.6
AODs	72	29	43	0	59.7
ROR	1570	116	677	777	85.5
LOI	796	81	417	298	83.7
COR	138	20	55	63	73.3
COI	92	13	58	21	81.6
COD	27	2	23	2	92
ASOR	64	8	28	28	77.7
Total	**4019**	**425**	**1911**	**1683**	**81.8**

Table 7
Results of applying mutation testing in all the applications.

Operator	Mutants				MS(%)
	generated	Alive	Killed	Equivalent	
OMCD	30	6	16	8	72
OOPD	31	0	29	2	100
OMCR	16	0	9	7	100
OOMU	25	0	8	17	100
OOMD	23	0	2	21	100
OOSS	4	0	4	0	100
OOSNR	5	0	5	0	100
OOSTR	57	0	39	18	100
OMNeT++ operators	**191**	**6**	**112**	**73**	**95**
SMCD	26	2	24	0	92
SMCR	25	0	25	0	100
SOMU	25	0	18	7	100
SOMD	14	0	6	8	100
SOSP	9	0	9	0	100
SOFNR	14	0	14	0	100
SOMAR	1	0	1	0	100
SOCIR	4	0	4	0	100
SIMCAN operators	**118**	**2**	**101**	**15**	**98**
MMCD	29	0	29	0	100
MMCR	56	0	56	0	100
MOMU	23	0	18	5	100
MOMD	27	0	20	7	100
MOSP	14	0	14	0	100
MOPIR	28	1	26	1	95
MOBLR	28	1	26	1	95
MPI operators	**205**	**2**	**189**	**14**	**99**
Total	**514**	**10**	**402**	**102**	**97.5**

The number of generated mutants using OMNeT and SIMCAN operators are similar. On the contrary, MPI operators produce a significantly greater number of mutants than OMNeT and SIMCAN operators. The reasons of this fact are two-fold. On the one hand, there are more appearances of MPI calls than OMNeT and SIMCAN calls. On the other hand, some of the MPI operators generate more mutants than the others, specially the *replace* and *shuffle* operators.

The case of *delete*, *replace* and *shuffle* operators, which generate the 60% of the total set of mutants, is particularly relevant. In the case of the mutant set generated by the SIMCAN and MPI operators, only 2 mutants were not killed,but they were identified as equivalent mutants. However, we obtain a slightly lower mutation score in those mutants generated by the OMNeT operators. This is due to the fact that this kind of operators have a lower impact in the selected applications, focused in general aspects of the application, which hampers the detection of these mutants.

On the contrary, *shift* mutation operators, which represent a 40% of the total set, generate mutants that have a low impact on the program

Table 8
Overview of the obtained results.

Operator	appCPU			appHPC			appPIPE		
	μ	%K	T	μ	%K	T	μ	%K	T
OOMU	1	100	24.9	1	100	5.9	1	75.6	70.1
OOMD	–	–	–	–	–	–	5.2	2.0	636.3
OMCD	–	–	–	–	–	–	5	10.4	473.2
OMCR	–	–	–	–	–	–	1.6	35.7	127.0
OOSNM	–	–	–	1	88.7	55.1	2.0	10	121.1
OOSTM	1	100	51.9	1	100	124.9	4.8	40.6	398.7
OOSR	1	100	66.6	1	100	53.6	2	9	112.0
OOPD	1	96.8	57.4	1	83	178	1.2	59.2	374.2
Avg	**1**	**99.2**	**50.2**	**1**	**94.34**	**83.48**	**3**	**30.3**	**326.6**
SOMU	1	100	1.2	1.4	60	458.6	1.7	44.8	150.1
SOMD	1	100	1.2	1.4	62.5	380.2	1.6	42	170.2
SOSP	1	100	1.2	1.5	50	736.9	2.7	54.7	236.8
SMCD	1	100	1.2	1.4	62.5	417.1	1.6	48	143.6
SMCR	1	100	10.8	1.5	54.6	433.1	1.7	46.7	135.7
SOCIM	–	–	–	–	–	–	1.5	50.5	296.7
SOFNR	1	100	51.0	1.8	50.0	157.7	2.6	54.7	197.3
SOMAR	–	–	–	1	100	116.7	–	–	–
Avg	**1**	**100**	**11.1**	**1.4**	**62.8**	**385.7**	**1.9**	**48.7**	**190**
MOMU	–	–	–	5.8	68.3	1706.2	3.6	42.9	311.1
MOMD	–	–	–	2.2	54.2	952.2	3 .7	32.4	309.5
MOSP	–	–	–	1.3	66.1	462.6	2 .1	47	153.5
MMCD	–	–	–	1.2	66.7	414.2	2 .9	41.6	233.9
MMCR	–	–	–	1.3	66.7	332.5	2.1	47	173.5
MOPIM	–	–	–	1.5	66.6	140.4	3	44.3	140.8
MOBLM	–	–	–	1.5	66.5	184.1	3	44.3	143.1
Avg	**–**	**–**	**–**	**2.1**	**65**	**598.8**	**2.9**	**42.7**	**209.3**
Total	**1**	**99.6**	**30.6**	**1.5**	**73.9**	**356**	**2.6**	**40.5**	**241.9**

execution and, in many cases, these mutants are considered equivalents. This fact is due to the event-oriented nature of the OMNeT++ platform, where the order of the sentences does not affect in the same way as the imperative programming. In addition, as it can be seen in the experiments, the average execution time of these mutants is slightly higher than the rest of mutants.

Additionally, we have analyzed the subsumption mutant relation. For this, we have implemented the subsuming mutants identification algorithm provided by Papadakis et al. (2016). The results obtained from the analysis of appCPU, appHPC and appPIPE show that 4.5%, 12% and 22.5% of mutants, respectively, are subsuming. Although this ratio seems to be low, let us remark that different studies focusing on the analysis of several applications (i.e. some of the most widely-used benchmark Unix utilities such as Grep, Sed, Flex, Make and Gzip) conclude that less than 5% of all the mutants are subsuming (Papadakis et al., 2016). However, 18 out of 23 proposed mutant operators have generated dominant (subsuming) mutants. Specifically, the mutation operators OOMU, OOSS, OMCR, OSNM, SORDMA only have generated subsumed mutants.

5.4.1. RQ1 - Effectiveness of our proposed mutation operators

In order to answer this question we generated and analyzed two different sets of mutants. One mutant set was generated by using our proposed mutation operators (see in Tables 1 and 2), while the other one was generated by using traditional mutation operators (see Table 6).

These results show that, during the testing process, a total number of 4019 mutants have been generated by using traditional operators, while only 514 mutants have been generated when our proposed mutation operators are applied. In this study, 1683 equivalent mutants (41.87%) have been generated by applying traditional mutation operators, while the mutant set generated by using our proposed operators only contains 102 equivalents (19.82%). In this latter case, TCE techniques have not detected duplicated mutants, which indicates the high quality of the generated mutant set. However, it is interesting to highlight that SIMCAN and MPI operators only produce 15 (12.71%) and 14 (6.82%) equivalent mutants, respectively.

The answer to **RQ1** is that our proposed mutation operators create a reduced set of mutants that has a high impact in the control flow of the application, which represents a substantial profit, both in terms of computational resources and time costs, in comparison with the traditional mutation operators that generate a vast number of mutants. This fact addresses one of the challenges of mutation testing, which is considered as a computational expensive technique. Hence, this solution is suitable to perform mutation testing over distributed applications in simulated environments.

5.4.2. RQ2 - Suitability of the obtained mutation score

Similarly to answer RQ1, we analyze our approach by performing the testing process using one test suite and two mutant sets. The first mutant set is generated by using our proposed mutation operators (see Table 7) and the second one contains mutants generated by using traditional mutation operators (see Table 6). It is important to remark that the same test suite has been used for both mutant sets.

In general terms, we appreciate that the testing process using a mutant set generated by applying our proposed mutation operators provides a better mutation score, on average (97.5%), than the testing process using a mutant set generated by using traditional operators (81.8%). If we analyze in detail the mutation score provided by executing the test suite over the first mutant set, we observe that, in the major part of the cases, we obtain a mutation score of 100 when our proposed operators are used. On the contrary, traditional operators provide poor results, which in this study ranges from 54.1 to 92.

The answer to **RQ2** is that the testing process provides promising results, in terms of mutation score, when a high-quality mutant set is used during the testing process. Since the obtained mutation score reaches 100 in the major part of the analyzed mutant set, we can state that this approach is appropriate to be applied in mutation testing. Moreover, we observe a correlation between the number of generated equivalents and the mutation score. Broadly speaking, a test suite executed over a mutant set generated by using a mutation operator that produces a reduced number of equivalents obtains a better mutation score that the same test suite executed over a mutant set generated by applying a mutation operator that generates a high number of equivalents.

5.4.3. RQ3 - Scalability of our approach

To answer this question we investigate the correlations between the complexity of the application under test and the obtained results in the testing process.

The answer to **RQ3** is that the effectiveness of each operator, in terms of execution time and the number of tests required to kill a mutant, depends on the complexity of the application under test. These factors have been analyzed for each application and the results are shown in Table 8, where μ represents the average number of tests required to kill a mutant, %K is the average rate of test cases that kill the mutant and T is the average time to kill the first mutant. In general terms, the more complex is the application under test, the higher is μ and the lower is %K.

The *delete, replace* and *shuffle* operators reach a good balance between the analyzed factors, that is, the average execution time of the generated mutants using these operators is usually lower than the mean and are prone to be killed by a higher quantity of test cases in comparison with *shift* operators.

6. Threats to validity

This section summarizes the threats to the validity of our study.

The main threat to internal validity is related to the manual steps applied during some phases of the testing process. The detection of equivalent mutants is an undecidable problem and, therefore, we have analyzed them by hand, which is an error-prone task. This means that

the results related to equivalence can present some errors. In addition, the applications may have dead zones in the source code and, although this factor may not affect the mutation score results, the mutants generated by injecting faults in dead zones are considered as equivalents. In order to alleviate this fact, we consider to include mechanisms based on control flow graph techniques to detect dead zones in the source code.

Another threat is related to the execution environment. The framework, illustrated in Section 4.1, consists of the simulation platform, the modeled applications and the mutation testing tool. The simulation platform has been used in several research papers, thus it is assumed to work in a proper way. In the same way, the experiments have been performed with applications modeled and validated by the authors in previous works (Merayo and Núñez, 2015; Núñez and Merayo, 2014). Consequently, these applications are considered to work correctly. Regarding MuTomVo, the tool that supports the framework, was intensively tested and analyzed before we have used it in our experiments.

Since the deterministic nature of the simulation platform allows to obtain the same results by launching the same experiments in different computational environments, all the experiments have been launched and executed in the same computer. Another threat is related to the selection of applications that can be considered as representative. We have selected three applications, with different sizes and levels of complexity, which represent some of the most popular computational paradigms. However, we cannot guarantee the selected applications are representative enough.

7. Related work

To the best of our knowledge, there does not exist any simulation framework of distributed systems that integrates testing techniques. Núñez and Hierons (2015) propose a methodology for validating cloud models integrating simulation and metamorphic testing techniques. Metamorphic relations are used to detect failures, such as performance or energy awareness, when simulating cloud. Rutherford et al. (2006) propose an approach that aims at the use of simulation to help with the selection of the most effective test adequacy criteria and the most effective test suite for the chosen criterion, among different adequate suites. In order to do it they apply traditional mutation operators to the simulation code and all the generated mutants are run against all the tests through an instrumented simulation. They use three distributed systems that are simulated in the SimJava simulation engine (Howell and Mcnab, 1998) to evaluate the proposal. The authors used MuJava (Ma et al., 2005) to generate mutants from the simulation code. When compared to this approach, our work contributes to the field of application of mutation in a different way. In our framework we propose the application of mutation testing techniques in simulation environments with the goal of checking simulation models. Our proposal is oriented to be used in simulation tools based on OMNeT++, that is a component-based C++ simulation library. On the contrary to the previous work, none of the available mutation testing frameworks for C++ (Delgado-Pérez et al., 2014; Kusano and Wang, 2013) are suitable to be applied in our framework. Delgado-Pérez et al. (2014) propose a set of class mutation operators for the object-oriented features of this language C++. Kusano and Wang (2013) apply mutation testing to C/C++ concurrent programs. Mutation is performed using their tool, CCMutator, that injects faults at concurrency constructs, such as semaphores, locks, mutual exclusion, etc. However, we are interested in the mutation of calls to the methods included in the OMNeT++ API, as well as in the SIMCAN and MPI APIs. Therefore, we have defined a set of specific mutation operators that introduce faults affecting these calls. We have also developed a new mutation tool, MuTomVo, to implement the new framework. We are not aware of any other work in the context of simulation of distributed systems that integrates mutation testing techniques.

Delamaro et. al presented a criterion for evaluating the adequacy of tests cases based on integration testing (Delamaro et al., 2001). This testing technique focuses on checking the correctness of modular software, incrementally testing the different modules that compose the system. Additionally, the authors of this work propose interface mutation, a technique that assesses the quality of how well the interfaces that communicates different modules have been tested. For this, a set of mutation operators are used to seed faults in the parts of the code that refer to interface communications, such as function calls, values returned from functions and global data sharing. Since our proposed mutation operators seed syntactical faults in function calls, the major part of them differs from the original interface mutation concept. First, those operators related with replacement and shifting operations are focused on injecting errors that represent wrong statement invocation and wrong statement placement, respectively. These errors affect to the program behavior without directly affecting to the connection between modules. Second, although some of the proposed mutation operators affect to the connection of the modules, such as scheduling time replacement and buffer length replacement, these have been designed ad-hoc to represent common errors committed by competent programmers, which were gathered from different sources. On the contrary, the interface-based mutation operators, which are based on C programming mistakes, are designed for a general purpose. Finally, none of the mutation operators proposed in this paper seed faults inside the called functions.

8. Conclusions

This paper presents a novel approach for checking the correctness of test suites focused on testing distributed applications. In particular, a framework called MuTomVo has been designed and implemented to perform the testing process over distributed scenarios. The SIMCAN simulation platform, aimed to model and simulate distributed systems, has been used to design these scenarios. In order to check the usefulness of MuTomVo, this framework has been implemented by a tool written in Java.

The main goal of this work is to facilitate the process of measuring the suitability of test suites for testing distributed applications. Since the execution of applications in distributed environments require a high cost, we use a simulation platform to represent the infrastructure where the application under test is executed. Thus, a complete set of mutation operators has been included in MuTomVo to inject different faults in the applications under test.

In order to check the suitability of MuTomVo we have carried out a thorough experimental study, where the mutation process has been performed over three applications executed in different distributed systems. In general, we have obtained promising results that show the usefulness of MuTomVo for testing different applications, that is, a client/server model, intensive computation and scientific pipelines.

The obtained results show that the proposed mutation operators are able to detect common mistakes produced by competent programmers. These injected errors generate unexpected behaviors in the applications under test, which are detected, in the major part of the cases, by the test suite. It is important to remark that there is a direct relation between the complexity of the application under test and the difficulty to detect errors. Hence, in simple applications like matrix multiplication, the major part of the mutants is detected by the entire test suite. However, in complex applications executed in highly distributed environments, these errors are only detected by a small number of test cases. Usually, complex applications have a complex input consisting of a large set of parameters. Thus, generating a test suite that covers all the combinations to produce all the possible behaviors in the application is unpractical due to the high computational cost.

In general, the testing process is costly. Basically, it requires high computational resources to execute a complete test suite consisting of a high number of tests, where each test is executed over a mutants set. Moreover, if this application is executed over a distributed system, this

cost significantly grows. Fortunately, this computational cost can be alleviated by using approaches like EMINENT (Cañizares et al., 2016), OUTRIDER (Cañizares et al., 2017) and TCE (Papadakis et al., 2015).

We observed that combining randomly generated test cases and manually generated test cases provides the best results, especially in complex applications that require a high number of parameters to configure a test case. While randomly generated test cases are enough to check simple applications, complex applications require a small intervention of the user in the testing process. In this experimental study, it was enough to provide 17 manually generated test cases.

Future work will include new mutation operators to cover a wider spectrum of errors. Thus, we expect to increase the efficiency of MuTomVo. Further analysis of different application will be studied. Additionally, we are planning to include parallel techniques to execute the complete mutation testing process in HPC systems. Finally, we will study the possibility of applying mutation testing techniques on hardware resources using simulation.

Acknowledgments

This work was supported by the Spanish MINECO/FEDER project DArDOS under Grants TIN2015-65845-C3-1-R and the Comunidad de Madrid project SICOMORo-CM under Grant S2013/ICE-3006. The first author is also supported by the Universidad Complutense de Madrid – Santander Universidades grant (CT17/17-CT18/17).

References

Althebyan, Q., Jararweh, Y., Yaseen, Q., AlQudah, O., Al-Ayyoub, M., 2015. Evaluating map reduce tasks scheduling algorithms over cloud computing infrastructure. Concurr. Comput. Pract. Exp. 27 (18), 5686–5699.

Ammann, P., Offutt, J., 2008. Introduction to Software Testing, First. Cambridge University Press, New York, NY, USA.

Arcuri, A., Iqbal, M.Z., Briand, L., 2012. Random testing: theoretical results and practical implications. IEEE Trans. Softw. Eng. 38 (2), 258–277.

Bucy, J.S., Schindler, J., Schlosser, S.W., Ganger, G.R., 2008. The DiskSim Simulation Environment Version 4.0 Reference Manual. Department Electrical and Computer Engineering, Carnegie Mellon University. Pittsburgh, PA 15213.

Cañizares, P.C., Merayo, M.G., Núñez, A., 2016. EMINENT: embarrassingly parallel mutation testing. Proceedings of the International Conference on Computational Science. pp. 63–73.

Cañizares, P.C., Núñez, A., de Lara, J., 2017. OUTRIDER: optimizing the mutation testing process in distributed environments. Proceedings of the International Conference on Computational Science. pp. 505–514.

Castañé, G.G., Núñez, A., Carretero, J., 2012. icancloud: a brief architecture overview. Proceedings of the International Symposium on Parallel and Distributed Processing with Applications. pp. 853–854.

Ciupa, I., Pretschner, A., Oriol, M., Leitner, A., Meyer, B., 2011. On the number and nature of faults found by random testing. Softw. Test. Verif. Reliab. 21 (1), 3–28.

Dean, J., Ghemawat, S., 2008. Mapreduce: simplified data processing on large clusters. Commun. ACM 51, 107–113.

Delamaro, M.E., Maidonado, J.C., Mathur, A.P., 2001. Interface mutation: an approach for integration testing. IEEE Trans. Softw. Eng. 27 (3), 228–247.

Delgado-Pérez, P., Medina-Bulo, I., Domínguez-Jiménez, J.J., García-Domínguez, A., Palomo-Lozano, F., 2014. Class mutation operators for c++ object-oriented systems. Ann. Telecommun. 70 (3), 137–148.

Delgado-Pérez, P., Segura, S., Medina-Bulo, I., 2017. Assessment of c++ object-oriented mutation operators: a selective mutation approach. Softw. Test. Verif. Reliab 27 (4–5), e1630.

DeMillo, R.A., Lipton, R.J., Sayward, F.G., 1978. Hints on test data selection: help for the practicing programmer. IEEE Comput. 11 (4), 34–41.

Deng, L., Offutt, J., Ammann, P., Mirzaei, N., 2017. Mutation operators for testing android apps. Inf. Softw. Technol. 81, 154–168.

DeSouza, J., Kuhn, B., Supinski, B.R., Samofalov, V., Zheltov, S., Bratanov, S., 2005. Automated, scalable debugging of MPI programs with intel® message checker. Proceedings of the International Workshop on Software Engineering for High Performance Computing System Applications. pp. 78–82.

Furnaghan, J., 2014. Oversim. The Overlay Simulation Framework. https://github.com/reines/oversim/. [Online; Latest commit on Jan 20, 2014. Accessed on Oct 15, 2017].

Gros, D., Alcidi, C., 2013. The Global Economy in 2030: Trends and Strategies for Europe. CEPS (Paperbacks), European Union.

Howell, F., Mcnab, R., 1998. simjava: a discrete event simulation library for java. Proceedings of the International Conference on Web-Based Modeling and Simulation. pp. 51–56.

Jackson, K., Ramakrishnan, L., Muriki, K., Canon, S., Cholia, S., Shalf, J., Wasserman, H., Wright, N., 2010. Performance analysis of high performance computing applications on the Amazon web services cloud. Proceedings of the International Conference on Cloud Computing Technology and Science. IEEE, pp. 159–168.

Jia, Y., Harman, M., 2011. An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. 37 (5), 649–678.

Kusano, M., Wang, C., 2013. Ccmutator: a mutation generator for concurrency constructs in multithreaded c/c++ applications. Proceedings of the International Conference on Automated Software Engineering. pp. 722–725.

Li, J., Humphrey, M., van Ingen, C., Agarwal, D., Jackson, K., Ryu, Y., 2010. eScience in the cloud: a modis satellite data reprojection and reduction pipeline in the windows azure platform. Proceedings of the International Symposium on Parallel Distributed Processing. pp. 1–10.

Ma, Y., Kim, S., 2016. Mutation testing cost reduction by clustering overlapped mutants. J. Syst. Softw. 115, 18–30.

Ma, Y., Offutt, A.J., Kwon, Y.R., 2005. Mujava: an automated class mutation system: research articles. Softw. Test. Verif. Reliab. 15 (2), 97–133.

Merayo, M.G., Núñez, A., 2015. Passive testing of communicating systems with timeouts. Inf. Softw. Technol. 64, 19–35.

Nanavati, J., Wu, F., Harman, M., Jia, Y., Krinke, J., 2015. Mutation testing of memory-related operators. Proceedings of the International Conference on Software Testing, Verification and Validation. pp. 1–10.

Núñez, A., Hierons, R.M., 2015. A methodology for validating cloud models using metamorphic testing. Ann. Telecommun. 70 (3–4), 127–135.

Núñez, A., 2011. New Contributions for Modeling and Simulating High Performance Computing Applications on Parallel and Distributed Architectures. Ph.D. thesis. Universidad Carlos III, Madrid, Spain.

Núñez, A., Fernández, J., Filgueira, R., García, F., Carretero, J., 2012. SIMCAN: a flexible, scalable and expandable simulation platform for modelling and simulating distributed architectures and applications. Simul. Modell. Pract. Theory 20 (1), 12–32.

Núñez, A., Merayo, M.G., 2014. A formal framework to analyze cost and performance in map-Reduce based applications. J. Comput. Sci. 5 (2), 106–118.

Offutt, A.J., King, K.N., 1987. A fortran 77 interpreter for mutation analysis. SIGPLAN Not. 22 (7), 177–188.

Papadakis, M., Henard, C., Harman, M., Jia, Y., Traon, Y.L., 2016. Threats to the validity of mutation-based test assessment. Proceedings of the International Symposium on Software Testing and Analysis. ACM, pp. 354–365.

Papadakis, M., Jia, Y., Harman, M., Traon, Y.L., 2015. Trivial compiler equivalence: a large scale empirical study of a simple, fast and effective equivalent mutant detection technique. Proceedings of the International Conference on Software Engineering. IEEE Press, pp. 936–946.

Piatov, D., Janes, A., Sillitti, A., Succi, G., 2012. Using the eclipse C/C++ development tooling as a robust, fully functional, actively maintained, open source C++ parser. Proceedings of the International Conference of Open Source Systems: Long-Term Sustainability. Springer, pp. 399–406.

Buyya, R., Calheiros, R.R., R., 2009. Modeling and simulation of scalable cloud computing environments and the Cloudsim toolkit: Challenges and opportunities.In: Proceedings of the High Performance Computing and Simulation Conference.

Ried, S., Kisker, H., Matzke, P., Bartels, A., Lisserman, M., 2011. Sizing the Cloud. Technical Report. Forrester.

Rutherford, M.J., Carzaniga, A., Wolf, A.L., 2006. Simulation-Based Testing of Distributed Systems. Technical Report CU-CS-1004-06. Department of Computer Science, University of Colorado.

Schad, J., 2010. Flying Yellow Elephant: Predictable and Efficient Mapreduce in the Cloud. Information Systems Group, Saarland University.

Szaszko, G., 2017. INET Framework for the OMNeT++ Discrete Event Simulator. https://github.com/inet-framework/inet. [Online; Latest commit on Oct 2, 2017. Accessed on Oct 15, 2017].

Varga, A., 2001. The OMNeT++ discrete event simulation system,. Proceedings of the European Simulation Multiconference.

Vesely, V., 2017. RINAsim. Recursive InterNetwork Architecture simulator. https://github.com/kvetak/RINA. [Online; Latest commit on Jun 22, 2017. Accessed on Oct 15, 2017].

Liu, W., and Fan, T., 2011. Live migration of virtual machine based on recovering system and cpu scheduling In: Proceedings of the Information Technology and Artificial Intelligence Conference, pp. 303-307.

Winsberg, E., 2010. Science in the Age of Computer Simulation. University of Chicago Press.

Wu, F., Nanavati, J., Harman, M., Jia, Y., Krinke, J., 2017. Memory mutation testing. Inf. Softw. Technol. 81, 97–111.

Pablo C. Cañizares is a Ph.D. Student that received the M.Sc. degree in Computer Science in 2015 at the Universidad Complutense de Madrid. He has published 10 papers in refereed journals and international venues. His research interests are formal methods, modelling and testing of distributed systems.

Alberto Núñez received the M.Sc. degree in computer science from Carlos III University of Madrid, Spain, in 2005 and the Ph.D. degree in computer science from the same university, in 2011. He is currently Assistant Professor with the Software Systems and Computation Department, Complutense University of Madrid, Spain. He has published more than 40 research papers in journals, books and national and international conferences. He regularly serves in the Program Committee of conferences such as SITIS, ICCCI or SAC-SOAP. His research interests include formal testing, performance analysis and modelling of cloud systems, especially on how to perform models and simulations. Dr. Núñez won the IBM Ph.D. Fellowship award in 2009.

Mercedes G. Merayo received her Ph.D. in Computer Science from Universidad Complutense de Madrid, Spain, in 2009. She holds an Associate Professor position in the Computer Systems and Computation Department at the same University. She has published more than 60 papers in refereed journals and international venues. She regularly serves in the Program Committee of conferences such as SEFM, ICTSS or QRS. Dr. Merayo has co-chaired QSIC 2011, SEFM 2013, ICTSS 2014 and SAC-SVT 2017 among others. Her current research interests include model based testing, distributed testing, asynchronous testing, mutation testing and timed extensions in formal testing.

7.4 EMINENT: EMbarrassINgly parallEl mutatioN Testing

7.4	
Publication data	
Authors:	Pablo C. Cañizares, Mercedes G. Merayo and Alberto Núñez
Title:	EMINENT: EMbarrassINgly parallEl mutatioN Testing
Publication:	16^{th} International Conference on Computational Science
Pub. Type:	Conference
Year:	2016
DOI/URL:	https://doi.org/10.1016/j.procs.2016.05.298
Pages:	11
CORE Ranking:	A
Contribution	
Summary:	In this paper we propose a dynamic distributed algorithm focused on HPC systems, called EMINENT, which has been designed to face the performance problems in mutation testing techniques. EMINENT alleviates the computational cost associated with this technique since it exploits parallelism in cluster systems to reduce the final execution time.
Technique:	Mutation Testing
Secondary techniques:	Parallel and Distributed Computing



Procedia Computer Science

Volume 80, 2016, Pages 63–73

ICCS 2016. The International Conference on Computational Science

EMINENT: EMbarrassINgly parallEl mutatioN Testing

Pablo C. Cañizares[1], Mercedes G. Merayo[1], and Alberto Núñez[1]

Universidad Complutense de Madrid, Madrid, Spain.
pablocc@ucm.es, mgmerayo@fdi.ucm.es, alberto.nunez@pdi.ucm.es

Abstract

During the last decade, the fast evolution in communication networks has facilitated the development of complex applications that manage vast amounts of data, like Big Data applications. Unfortunately, the high complexity of these applications hampers the testing process. Moreover, generating adequate test suites to properly check these applications is a challenging task due to the elevated number of potential test cases. *Mutation testing* is a valuable technique to measure the quality of the selected test suite that can be used to overcome this difficulty. However, one of the main drawbacks of mutation testing lies on the high computational cost associated to this process.

In this paper we propose a dynamic distributed algorithm focused on HPC systems, called `EMINENT`, which has been designed to face the performance problems in mutation testing techniques. `EMINENT` alleviates the computational cost associated with this technique since it exploits parallelism in cluster systems to reduce the final execution time. In addition, several experiments have been carried out on three applications in order to analyse the scalability and performance of `EMINENT`. The results show that `EMINENT` provides an increase in the speed-up in most scenarios.

Keywords: Mutation testing, Scientific Computing, Parallel and Distributed Computing

1 Introduction

During the last years, the emergence of new technological trends, such as next-generation networks, always-connected mobile broadband and media services, has facilitated the rising of a new generation of IT. This emergence is characterised by high-speed and high-connectivity connection networks simultaneously used by millions of users. Similarly, there has been a rise of a new generation of applications and services, such as social networks and instant messaging applications, that allows users to share and process images, send posts and communicate with other users in a fast and accessible way. Thus, the growing popularity of these services, has lead to a massive data generation. For instance, in 60 seconds, WhatsApp users share 490,320 messages, Instagram users filter 216,000 images, Twitter users send 347,222 tweets and Facebook users share 2,246,000 posts [24]. In order to handle and process this massive amount of data, current systems have to face the challenge of performing these techniques efficiently and effectively.

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

doi:10.1016/j.procs.2016.05.298

Hence, it is important that communication networks achieve a high bandwidth with a low latency to transfer large amounts of data, while computational resources are exploited in parallel. One of the most used solutions to reduce long execution time is *High Performance Computing* (in short, HPC), a computational solution which provides an excellent price-performance ratio. The importance of this paradigm is reflected in the TOP-500 list, which shows that the 500 most powerful computers in the world are clusters [14].

Another important aspect that must be taken into account is the complexity of these services that are composed of diverse processes, such as text compression and image filtering. This fact hampers the validation process. Nevertheless, it is necessary to build adequate test suites to check their correctness before deploying them in the production environment. Currently, *testing* is the most often used technique to check the validity of software. One of the main difficulties in applying testing methodologies is to design an *appropriate* test suite. Mutation testing allows to improve the design of high quality test suites. This technique is based on applying *mutation operators* to programs that make small syntactic changes in order to produce a set of *mutants*. The idea is that if a test suite is able to distinguish between a program and the generated mutants, it should be good at detecting a faulty implementation. It helps to determine the effectiveness of a test suite and helps during the test generation. The effectiveness of a test suite is established on the basis of how many of the mutants it distinguishes from the original program. However, mutation testing is computationally expensive, since the number of mutants that are generated is huge and they must be executed against the test suite. In consequence, high computational power is required to speed-up the mutation testing process.

In this paper we propose `EMINENT`, a scalable, dynamic, and HPC-oriented algorithm [17, 15, 2], based on embarrassingly parallel computation ideas [9, 4]. `EMINENT` focuses on reducing the execution time associated to the classical mutation testing scheme. The proposed algorithm is *scalable*, the overall system performance increases as the computational resources. It is also *dynamic*, since the processed workload of each computational resource is assigned depending on its underlying characteristics. Moreover, `EMINENT` is focused on HPC and uses the shared resources of cluster systems to solve the same computational problem. This approach has been implemented using MPI, a standard Message-Passing Interface library to improve the communications in high performance environments, which properly fits with dynamic distribution strategies [5, 25].

The rest of the paper is structured as follows. Section 2 presents the state of the art of parallel mutation testing. Next, in Section 3 we describe `EMINENT` in detail. Section 4 presents some performance experiments by using three different applications. Finally, in Section 5, we present the conclusions and some lines of future work.

2 State of the art

Currently, several cost reduction techniques to improve the execution time of mutation testing can be found in the literature. These techniques are traditionally divided into three approaches: *do fewer*, *do faster* and *do smarter*. The proposal presented in this paper focuses on parallel testing, which is classified in the *do faster* approach. Although we have found some works in this research field during the last decades, it is worth mentioning that most of the proposals were introduced during the early nineties and the last 4 years.

The first contribution in parallel mutation testing can be traced back to 1988 with Mathur and Krauser [13]. In their approach, they proposed a novel technique to reduce execution costs using a vector processor. In this work, multiple mutants are simultaneously executed in a single processor using a sequence of vector instructions. Even though the approach greatly

increases the computational performance of the mutation testing scheme, it is limited to mutants generated with scalar variable replacement operator. Afterwards, authors extended their work with a high performance approach based on shared-memory, called mutation unification, to support several existing mutation operators [11, 21]. In their studies, compilation was identified as a major bottleneck of the scheme. However, this issue can be alleviated by using current techniques that can be found in the literature [26, 12].

There exist multiple mutation testing frameworks that include parallel techniques to improve the performance and, consequently, to reduce the computational cost [10, 23]. Despite the benefits obtained by the use of Single Instruction Multiple Data improvements, these systems are limited by the number of processors that are comprised. Hence, it is necessary to include new distributed schemes of mutation testing that address this scalability issue.

In order to alleviate this problem, Offut et al. proposed the first mutation testing approach based on Multiple Instruction Multiple Data systems [16]. This work presents a parallel interpreter, called *HyperMothra*, which was implemented on a sixteen processor Intel iPSC/2 hypercube. In addition, diverse static schemes of distribution algorithms are included, such as distributing mutants in original order and distributing mutants randomly and uniformly by mutation type. The authors stated that the performance achieves almost a linear speedup over Mothra's sequential interpreter but they also identified the communications as the bottleneck of the system.

In the same line, Byoungju and Mathur presented the $P^M othra$ system [3]. This approach has a flexible architecture designed to provide a high degree of scalability. The system also provides the tester with a transparent interface to a distributed machine and includes a dynamic distribution algorithm that serves mutants to the available nodes. As in the previous proposals, the communication network is a bottleneck and slows-down the performance of the system.

Most recently, Mateo and Usaola have presented a study for adapting the existing cost reduction techniques to current technologies [20]. They introduce $Bacterio^P$, a parallel extension of the mutation testing tool *Bacterio* [19], using *Java-RMI* [6] in order to communicate the nodes of the network. In addition, the authors include five distribution schemes using dynamic and static distributions. Among these schemes, it is worth noting the *Parallel Execution with Dynamic Ranking and Ordering (PEDRO)* algorithm. This contribution is a dynamic distribution algorithm based on *Factoring Self-Schedulling* ideas [8], that considers to address the well known communication efficiency problem. Although this proposal achieves better results than the previous works, the mechanisms used in the communications are not the most adequate for high performance environments due to the high latency introduced by this technology. It has been shown that Java-RMI is 3 to 5 times slower than MPI [18]. Hence, we consider that the distribution process can be improved in order to achieve a higher level of parallelism by increasing resources efficiency.

In 2014, Saleh and Nagi presented the *HadoopMutator* framework. It is based on Map-Reduce programming model and distributes and executes mutant generation and testing process [22]. The framework is based on *Hadoop* engine and *Pitest* mutation testing framework. This approach follows a static schema in which the inclusion of dynamic distribution algorithms is not considered. Thus, this framework is not oriented to heterogeneous and dynamic environments.

3 EMINENT

Nowadays, there exist several techniques to improve the performance of the mutation testing process. In this paper we propose `EMINENT`, an algorithm to distribute the workload of this

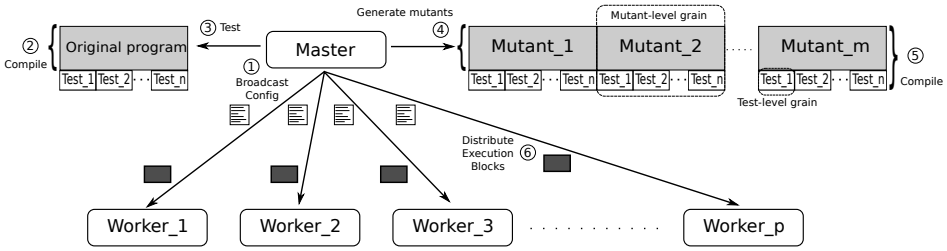


Figure 1: General scheme of EMINENT

process in order to reduce the execution time. The main goal of this work is to achieve an *scalable*, *dynamic* and *high performance* solution to face the computational challenges associated with mutation testing. Next, we describe the main features of EMINENT.

- *Scalable*: EMINENT has been designed to be deployed and executed in a distributed system. The increment in the quantity and quality of system's resources means the increase in its computational performance. EMINENT presents two types of scales: *horizontal and vertical*. The former allows to include more computing nodes to the system, while the latter allows to extend the computational resources in each node.

- *Dynamic*: In order to maximise the exploitation of computational resources, EMINENT splits the input dataset into blocks and dynamically delivers them to the available CPUs. Once a node finishes the execution of a block, it is provided with a new block until all the blocks have been processed. This distribution scheme benefits heterogeneous systems.

- *High performance*: The proposed algorithm is based on a *high performance* schema in which the shared resources of several machines are used as a whole to perform the mutation testing process. The testing process is executed in parallel over all the nodes of a cluster, taking advantage of the low latency communication network to maximise the parallelism and enhance the overall performance.

Algorithm 1 presents the main steps of EMINENT. The proposed scheme uses different processes. On the one hand the *master* process, that is responsible for orchestrating the algorithm. It splits the workload of the testing process in *execution blocks* and distributes them among the worker processes. On the other hand, the worker processes execute them and send the results back to the master process. The number of workers processes that are instantiated in the algorithm is variable and can be defined by the user.

Figure 1 shows the basic scheme of EMINENT. The first step consists in the selection of both the *source code* of the program to which the mutation testing process will be applied and the *test suite* that will be used during this process. Then, the master process compiles the original program ② and, if the compilation finishes successfully, the *testing process* begins. At this point, the master executes all the test cases in the selected test suite and stores the results ③. If the execution of all test cases is correct, the master invokes an external mutation testing tool to generate ④ and compile ⑤ all the program mutants. Mutants are produced by using *mutation operators* that aim to simulate common faults. Each mutation operator makes a small syntactic change in the source code. The execution of the generated mutants is distributed by

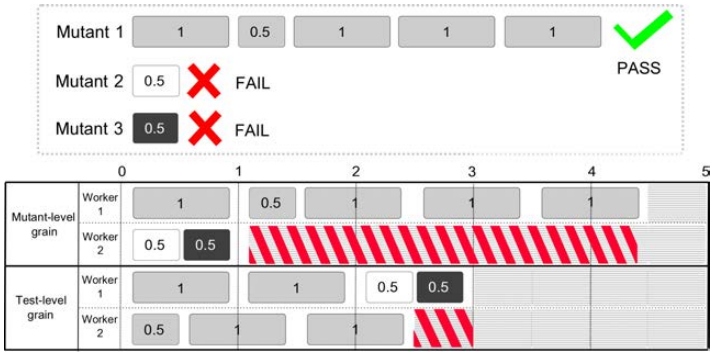


Figure 2: Comparison of different grain size (mutant vs test)

the master process among the the workers ⑥. For each mutant, the master process dispatches the test cases to the worker processes that will execute them against the mutant. The obtained results are sent to the master, that compares them with the ones produced by the original program. In the case that a difference is detected, the mutant will be considered *killed*, and all the running executions associated with it will be aborted and no more tests will be executed against it. The process continues until all the test cases are executed against all the mutants. Finally, the master process calculates the *mutation score* of the process, that indicates the percentage of killed mutants over the total number of mutants.

`EMINENT` uses a *test-level grain*, where the *execution blocks* are composed by a single test case, in contrast to the *mutant-level grain*, where the *execution blocks* are composed by the complete test suite, that usually is implemented in the dynamic distribution approaches existing in the literature. Figure 2 illustrates the difference between both of them when applied in the execution of mutant against a set of test.

Each algorithm is executed using 2 worker processes to test 3 different mutants against a test suite that contains 5 tests cases. The mutant 1 passes all the test cases in 4.5 units of time ($t_{c1} = 1, t_{c2} = 0.5, ..., t_{c5} = 1$) and mutants 2 and 3 fail the first test in 0.5 units of time. On the one hand, in the mutant-level grain algorithm the first worker executes all the tests over the mutant 1. Then, in parallel, the second worker executes, consecutively, the test 1 over the mutant 2 and 3. The total time elapsed during this process is 4.5 units of time. On the other hand, in the test-level grain scheme, the test cases can be executed by any of the workers. First, mutant 1 is executed against all the test cases. Then, when no more test cases have to be applied to mutant 1, worker 1 executes test 1 on mutants 2 and 3. In this case, the total time elapsed is 3 units of time. It is important to emphasize that the worker 2 is idle from $t = 1$ to $t = 4.5$ when using the mutant-grain level. However, this worker is only idle from $t = 2.5$ to $t = 3$ when the test-grain level is used. This difference means that the test-grain level is more adaptable to heterogeneous environments and allows to maximise the resources usage. As a consequence, the overall time of the testing process is reduced.

Algorithm 1 Parallel Mutation Testing

Require: *config*

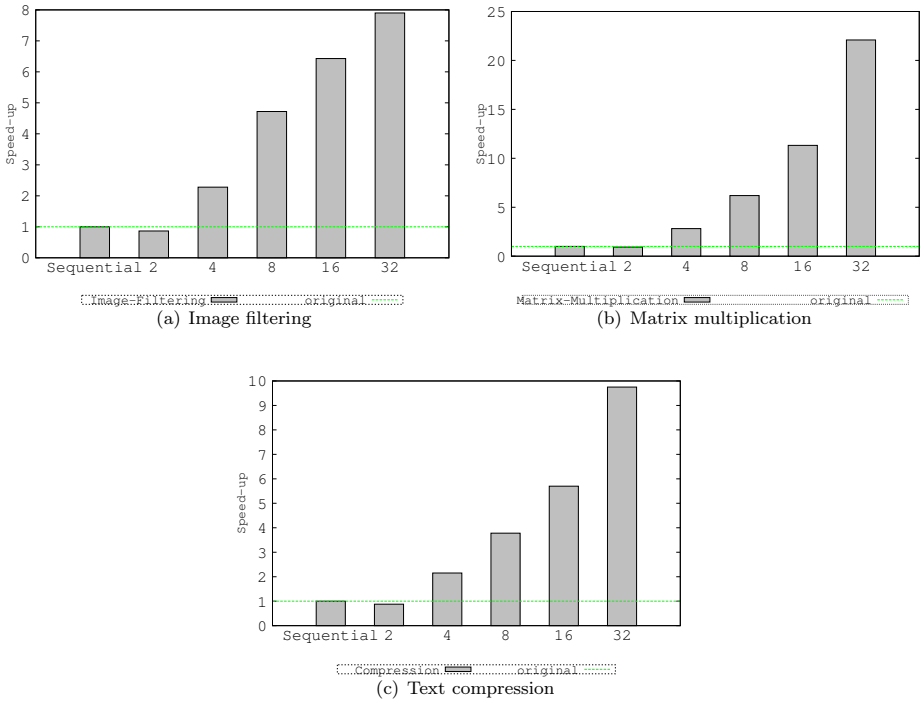
```
1: MPI_Init();
2: numprocs ← MPI_Comm_size();
3: myId ← MPI_Comm_rank();
   // Master process
4: if (myId == MASTER) then
5:   originalResults ← executeTests(originalProgram, config.getTests());
6:   if areResultsCorrect(originalResults) then
7:     mutantList ← generateMutants();
8:     compileMutantsAndTests();
9:     MPI_Bcast (Config, MASTER);
10:    resultsMutants ← builtResultsTable (mutantList, config.getTests()); // Initial distribution among worker processes
11:    while (i<numProcs and getRemainingMutants (resultsMutants)>0) do
12:      currentMutant ← getCurrentMutant(resultsMutants);
13:      currentTest ← getCurrentTest (currentMutant, resultsMutants);
14:      execBlock ← buildExecBlock (currentMutant, currentTest, config.getExecBlock());
15:      MPI_Send (execBlock, i);
16:    end while// While there are remaining mutants ...
17:    while (continueProcessing) do
18:      result ← MPI_Recv (ANY, status);
19:      continueProcessing ← updateResults (result, resultsMutants);
20:      if (getRemainingMutants (resultsMutants)>0) then
21:        currentMutant ← getCurrentMutant(resultsMutants);
22:        currentTest ← getCurrentTest (currentMutant, resultsMutants);
23:        execBlock ← buildExecBlock (currentMutant, currentTest, config.getExecBlock());
24:        MPI_Send (execBlock, status.MPI_SENDER);
25:      end if
26:    end while
27:  end if
     // Worker process
28: else
29:  while (continueProcessing) do
30:    MPI_Recv (execBlock, MASTER);
31:    result ← execute (execBlock);
32:    MPI_send (MASTER, result);
33:    continueProcessing ← execBlock.continueProcessing;
34:  end while
35: end if
```

4 Experiments

In this section we present some experiments to check the scalability and performance of `EMINENT`. We have used Milu [10], a well known mutation testing tool for the generation of mutants.

The experiments have been performed in a cluster that consists of 8 nodes interconnected through a Gigabit Ethernet network. Each node contains a Quad-Core Intel(R) Core(R) i5-3470

CPU at 3.4 Ghz with hyper-threading, 8 GB of RAM and 500GB HDD. In order to measure the scalability of the proposed algorithm, we have performed several executions with different number of processors.



(a) Image filtering

(b) Matrix multiplication

(c) Text compression

Figure 3: Speed-up of the applications using `EMINENT`

We have tested three different applications: *image filtering*, *massive computational application* and *text compression*. The first one applies filters to images. It uses 3 different filters to process images: *grayscale*, *median* and *saturation*. Initially, all the images are located in a database node with a total size of 2,5 GB. The master reads the image dataset from a remote database and distributes them among the worker processes using the communication network. Then, these images are filtered by the workers and saved in the local storage to check the I/O scalability. In this case, a test is given as a tuple $T =< I, F, O, C_{md5} >$ where I is the image to be processed, F is the filter to be applied and O is the filtered image. Finally, C_{md5} is a function that calculates the md5 hash of O. This value is used to compare the results obtained from the application of the test to the original program and each one of the mutants. This application and mutants were executed against 3200 test cases. The number of mutants generated to measure the test suite effectiveness was 250.

The second application performs a large number of operations in order to multiply matrixes, which means a huge computational load. The experiments performed over this application intended to analyse the computational scalability of the proposed algorithm. In this case a test

is given as a tuple $T =< R, N, O, C_{md5} >$ where R is the seed used to build a pseudo-random matrix, N is the size of the matrixes, O is the result of the matrix multiplication and C_{md5}, as in the previous case, is a function that calculates the md5 hash of O. The test suite used to test this application contained 2000 tests and the number of mutants generated was 100.

The last application performs file compression using the *LZ4* algorithm. Initially, all the text files are stored in a remote repository composed of 1500 elements with a total size of 3 GB. Then, these files are compressed by the workers. The test cases are given as tuples $T =< F, O, C_{md5} >$ where F is the file to be processed, O is the result of the compression and C_{md5} is a function which calculates the md5 hash of O. In this case the test suite contained 1000 tests and a total of 200 mutants were generated.

Figure 3 shows the overall speed-up obtained by using `EMINENT` with these applications. The performance improvement is measured on the basis of a sequential execution. The applications were executed using 2, 4, 8, 16 and 32 processes. The greater the number of processes the greater the speed-up is. In all the cases, one of the processes acts as the master, which distributes the workload among the rest of the processes, which play the role of workers. Thus, the master process is not involved in the execution of the test cases. It is worth noting that the reduction observed in the performance during the experiments carried out with 2 processes with respect to the obtained during sequential execution, is due to the time spent in the communication between the master and worker processes.

Figure 3(a) shows the overall speed-up obtained during the testing of the *image filtering* application. The maximum achieved speed-up is close to 8. This performance reduction is because of the high volume of network and I/O traffic generated by this application, which acts as a system bottleneck in the database node. Since the master process does not execute tests, there is a significant increasing of performance when 2 and 4 processes are used, which means that 1 and 3 worker processes are executed, respectively. In this case, using 4 processes obtains a performance 2,85 greater than using 2 processes. By the contrary, the experiments using 4 to 32 processes show that the obtained performance slowly increases when the number of worker processes increases. This is due to the increase in the network traffic generated by obtaining the images from the database, which is shared by all the worker processes.

Figure 3(b) shows the results obtained in the experiments performed on the *massive computational* application. In this case the maximum achieved speed-up is 22. This fact is due to this application executes most of the operations in the CPU and sends small amounts of data between processes, which reduces the communication bottleneck. Consequently, the obtained performance when the number of processes varies from 4 to 32 increases almost linearly. Figure 3(c) shows the results corresponding to the *text compression* application. The maximum achieved speed-up is close to 10. Similarly to the image filtering application, the performance seems to be limited by the vast quantity of network traffic generated to obtain the text files from the remote repository.

With the aim of measuring the effectiveness of `EMINENT` we have compared it with a previous proposal to improve the performance of the mutation testing scheme, PEDRO algorithm. This approach presents the best results if we compare it with other works that have dealt with this issue. In order to carry out the comparison between these two algorithms, two types of execution grain were used. The PEDRO algorithm is based on mutant-level grain and `EMINENT` is based on test-level grain. We use the same applications but in this case the test suite generated for each of them contained 5000 test cases. The size of the data repositories used in the image filtering and text compression applications was increased up to 4 GB and 6 GB respectively. Figure 4 shows a comparative chart of the results obtained from both algorithms. All the experiments were performed with 32 processes in order to to maximise the parallelism level. In addition,

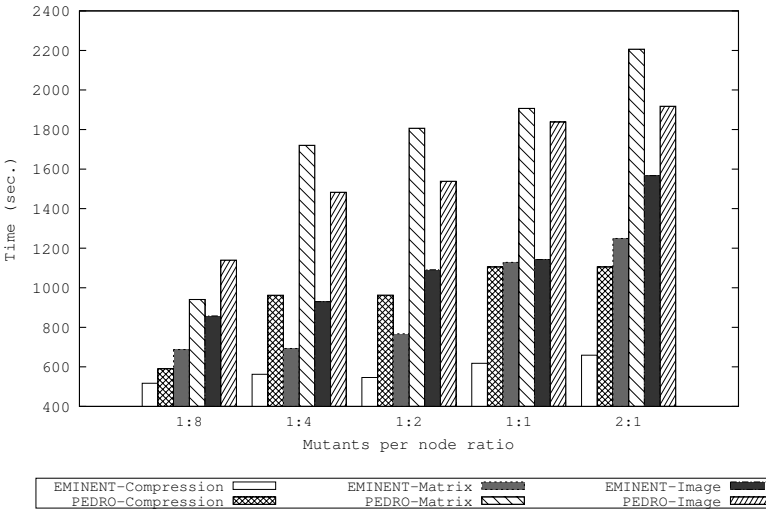


Figure 4: Results of comparison between EMINENT vs PEDRO

the executions were performed with different mutants per node ratio to check the scalability of both algorithms under different workload conditions. This ratio ranges from 1 mutant for each 8 nodes, to 2 mutants for each node. The chart shows that the test-level grain of EMINENT is more adaptable and suitable to maximise the exploitation of the computational resources than the mutant-level grain of PEDRO. In all the experiments carried out with EMINENT the performance obtained was better than the one showed by PEDRO. Overall, EMINENT scales better than PEDRO due to the execution time seems to grow slower in the experiments performed with EMINENT when the mutants per node ratio increases.

5 Conclusions and future work

We have presented a distributed algorithm, called EMINENT, designed to face the computational challenges associated with mutation testing. The main goal of this algorithm is to provide a scalable, dynamic and HPC-based method for reducing the execution cost by maximizing parallelism in this testing technique. The evaluation results show that EMINENT provides a better speed-up than the existing approaches in three different applications. These experiments have shown that EMINENT has a high adaptability level in heterogeneous computational environments where different usage levels of CPU, I/O and network are considered. Moreover, we have shown that the overall performance of the system systematically increases with the number of processes. This fact is a clear indicator of the scalability of our proposal.

As future work we plan to extend our proposal in order to parallelise the testing process of the original program so that we can reduce even more the time costs. Moreover, we will integrate other HPC techniques, such as an online compression layer, to reduce the loss of performance introduced by the network latency. Finally, we will adapt to our framework existing formal approaches to test distributed and timed systems [1, 7].

Acknowledgements

Research partially supported by the Spanish MEC project DArDOS (TIN2015-65845-C3-1-R) and the Comunidad de Madrid project SICOMORo-CM (S2013/ICE-3006).

References

[1] C. Andrés, M.G. Merayo, and M. Núñez. Formal passive testing of timed systems: Theory and tools. *Software Testing, Verification and Reliability*, 22(6):365–405, 2012.

[2] P.C. Cañizares, A. Núñez, M. Núñez, and J.J. Pardo. A methodology for designing energy-aware systems for computational science. In *Proceedings of the International Conference on Computational Science*, volume 51, pages 2804–2808. Elsevier, 2015.

[3] B. Choi and A.P. Mathur. High-performance mutation testing. *Journal of Systems and Software*, 20(2):135–152, 1993.

[4] B. Fjukstad, J.M Bjørndalen, and O. Anshus. Embarrassingly distributed computing for symbiotic weather forecasts. In *Proceedings of the International Conference on Computational Science*, volume 18, pages 1217–1225. Elsevier, 2013.

[5] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, 09 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[6] W. Grosso. *Java RMI*. O'Reilly & Associates, Inc., 1st edition, 2001.

[7] R.M. Hierons, M.G. Merayo, and M. Núñez. Implementation relations and test generation for systems with distributed interfaces. *Distributed Computing*, 25(1):35–62, 2012.

[8] S.F. Hummel, E. Schonberg, and L.E. Flynn. Factoring: A method for scheduling parallel loops. *Journal of Communications of ACM*, 35(8):90–101, 1992.

[9] L. Ismail and L. Khan. Implementation and performance evaluation of a scheduling algorithm for divisible load parallel applications in a cloud computing environment. *Software: Practice and Experience*, 45(6):765–781, 2015.

[10] Y. Jia and M. Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language. In *Proceedings of Practice and Research Techniques*, pages 94–98. IEEE, 2008.

[11] E.W. Krauser, A.P. Mathur, and Vernon J. Rego. High performance software testing on simd machines. *IEEE Transactions on Software Engineering*, 17(5):403–423, 1991.

[12] Y. Ma, A.J. Offutt, and Y. Kwon. Mujava: An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[13] A.P. Mathur and E.W. Krauser. Modeling mutation and a vector processor. In *Proceedings of the International Conference on Software Engineering*, pages 154–161, 1988.

[14] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst D. Simon. Top500 supercomputer sites, 2016. http://www.top500.org.

[15] A. Núñez, R. Filgueira, and M.G. Merayo. Sancomsim: A scalable, adaptive and non-intrusive framework to optimize performance in computational science applications. In *Proceedings of the International Conference on Computational Science*, pages 230–239. Elsevier, 2013.

[16] A.J. Offutt, R.P. Pargas, S.V. Fichter, and P.K. Khambekar. Mutation testing of software using a mimd computer. In *Proceedings of the International Conference on Parallel Processing*, pages 255–266, 1992.

[17] D.R. Penas, P. González, J.A. Egea, J. R. Banga, and R. Doallo. Parallel metaheuristics in computational biology: An asynchronous cooperative enhanced scatter search method. In *Proceedings of the International Conference on Computational Science*, pages 630–639, 2015.

[18] K. Qureshi and H. Rashid. A performance evaluation of rpc, java rmi, mpi and pvm. *Malaysian Journal of Computer Science*, 18(2):38–44, 2005.

[19] P. Reales and M. Polo. Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases. In *Proceedings of the International Conference on Software Maintenance*, pages 646–649. IEEE, 2012.

[20] P. Reales and M. Polo. Parallel mutation testing. *Journal of Software Testing, Verification and Reliability*, 23(4):315–350, 2013.

[21] V. Rego and A.P. Mathur. Concurrency enhancement through program unification: a performance analysis. *Journal of Parallel and Distributed Computing*, 8(3):201–217, 1990.

[22] I. Saleh and K. Nagi. Hadoopmutator: A cloud-based mutation testing framework. In *Software Reuse for Dynamic Systems in the Cloud and Beyond*, pages 172–187. Springer, 2014.

[23] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for java. In *Proceedings of the International Conference of Software Engineering Conference and the ACM SIGSOFT Symposium*, pages 297–298. ACM, 2009.

[24] Statista. Media usage in an internet minute as of august 2015, 2015. Available at `http://www.statista.com/statistics/195140/new-user-generated-content-uploaded-by-users-per-minute/`.

[25] M. Towara, M. Schanen, and U. Naumann. Mpi-parallel discrete adjoint openfoam. In *Proceedings of the International Conference on Computational Science*, volume 51, pages 19–28. Elsevier, 2015.

[26] R.H. Untch, A.J. Offutt, and M.J. Harrold. Mutation analysis using mutant schemata. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 139–148, 1993.

7.5 OUTRIDER: Optimizing the mUtation Testing pRocess In Distributed EnviRonments

7.5	
Publication data	
Authors:	Pablo C. Cañizares, Alberto Núñez and Juan de Lara
Title:	OUTRIDER: Optimizing the mUtation Testing pRocess In Distributed EnviRonments
Publication:	17^{th} International Conference on Computational Science
Pub. Type:	Conference
Year:	2017
DOI/URL:	`https://doi.org/10.1016/j.procs.2017.05.095`
Pages:	10
CORE Ranking:	A
Contribution	
Summary:	In this work we propose an HPC-based optimization that contributes to bridging the gap between the high computational cost of mutation testing and the parallel infrastructures of HPC systems aimed to speed-up the execution of computational applications. This optimization is based on our previous work called EMINENT, an algorithm focused on parallelizing the mutation testing process using MPI. However, since EMINENT does not efficiently exploits the computational resources in HPC systems, we propose 4 different strategies to alleviate this issue.
Technique:	Mutation Testing
Secondary techniques:	Parallel and Distributed Computing

OUTRIDER: Optimizing the mUtation Testing pRocess In Distributed EnviRonments

Pablo C. Cañizares[1], Alberto Núñez[1], and Juan de Lara[2]

[1] Universidad Complutense de Madrid, Madrid, Spain.
[2] Universidad Autónoma de Madrid, Madrid, Spain.

Abstract

The adoption of commodity clusters has been widely extended due to its cost-effectiveness and the evolution of networks. These systems can be used to reduce the long execution time of applications that require a vast amount of computational resources, and especially of those techniques that are usually deployed in centralized environments, like testing. Currently, one the main challenges in testing is to obtain an appropriate test suite. Mutation testing is a widely used technique aimed to generate high quality test suites. However, this technique requires a high computational cost to be executed.

In this work we propose an HPC-based optimization that contributes to bridging the gap between the high computational cost of mutation testing and the parallel infrastructures of HPC systems aimed to speed-up the execution of computational applications. This optimization is based on our previous work called `EMINENT`, an algorithm focused on parallelizing the mutation testing process using MPI. However, since `EMINENT` does not efficiently exploits the computational resources in HPC systems, we propose 4 different strategies to alleviate this issue. Moreover, a thorough experimental study has been carried out by using different applications to analyze the scalability and performance obtained with `OUTRIDER`.

Keywords: Parallel and Distributed Computing, High Performance Computing, Mutation testing

1 Introduction

Wired communications have experienced a notorious growth with the use of fiber optic, reaching in experimental environments a speed of 560 Gbit/s [10]. The main features of these networks, such as low latency and very high bandwidth, have made commodity clusters a cost-effective solution, it being the main source for high performance computing (in short, HPC). As an example, in the most recent survey of the fastest 500 computers in the world [12], 86.4% are clusters. This trend has allowed several research advances in scientific computing by using HPC techniques [16, 1]. Moreover, techniques that require a long execution time and are usually executed in centralized environments, like testing, can be deployed in clusters in order to reduce its high computational cost [2].

At present, testing is one of the most extended mechanisms to check the correctness of software [13]. However, in order to properly check the validity of a program, it is required to generate an appropriate test suite (in short, TS), which in most cases is a difficult and challenging task. Moreover, a large TS requires a very long execution time. Fortunately, there exist mechanisms, like mutation testing (in short, MT), focusing on improving the design of high quality TS. Basically, MT is based on applying mutation operators to programs that make small syntactic changes in order to produce a set of mutants. The idea is that if a TS is able to distinguish between a program and the generated mutants, it should be good at detecting a faulty implementation. Thus, the effectiveness of a TS is established on the basis of calculating the number of mutants that are distinguished from the original program.

In recent years, MT has been successfully applied in several fields [8, 4]. However, MT is computationally expensive because a large TS is executed over a vast collection of mutants. It is therefore required high computational resources in order to speed-up the testing process.

In this paper we present `OUTRIDER`, an HPC-based optimization to improve the overall performance of the MT process. Our approach uses the `EMINENT` algorithm as basis [2], which focuses on reducing the execution time of MT by parallelizing the testing process in HPC systems. However, since `EMINENT` does not properly exploit the resource usage in HPC systems, we use our proposed optimization consisting of 4 different strategies to alleviate this issue. These strategies are summarized as follows:

- Parallelizing the execution of the TS over the original application. While existing works in the literature sequentially execute the TS over the original application, we propose to distribute the test cases among different processes to be executed in parallel.
- Sorting the TS by using the execution time of each test case as sorting criteria. Since the TS is executed over the original application in the initial phase of the testing process, the time required to execute each test case can be collected. In the next phase of the testing process, where the TS is executed over each mutant, this information can be used to specify the order of execution for each test case. Hence, the test case requiring the shortest amount of time to be executed is processed in the first place. In general, for each test case, the shorter execution time is required, the greater priority to be executed.
- Enhancing the test case distribution. This strategy focuses on maximizing the number of different mutants executed in parallel. We say that a mutant is executed in parallel when different test cases are executed over this mutant, in different processes, at the same time. In this case, the resource usage efficiency may decrease if the test case that kills the mutant is executed in parallel with other test cases. Consequently, the execution of those test cases that do not kill the mutant is useless for obtaining the final results and, therefore, the computational resources are not efficiently used.
- Categorizing cloned and equivalent mutants. This strategy is based on grouping those mutants that are clones and equivalents. We say that two different mutants are clones when the resulting executable files obtained from their compilation are identical. A mutant is considered equivalent with respect to the original application when there is no test case that kills this mutant. The main goal of this strategy consists in avoiding the complete execution of both equivalent and cloned mutants.

The remainder of this paper is organized as follows. Section 2 presents the state of the art. Section 3 describes `OUTRIDER` in detail. Next, in Section 4, we present some performance experiments by analyzing the suitability of the proposed optimization. Finally, in Section 5, we present our conclusions and future work.

2 State of the art

Since the first contributions in MT, there has been a constant effort to alleviate its high computational cost. As a result, different works aimed to improve the performance of the MT process can be found in the literature. These contributions can be classified in two main groups.

The first group focuses on reducing the total number of mutants without losing a significant effectiveness. Among them, it is specially relevant *mutant sampling*, a technique based on randomly selecting a subset of mutants [22], *mutant clustering*, which selects a collection of mutants by using clustering algorithms [11] and *high order mutation*, that generates a reduced set of mutants, which are created by applying multiple mutation operators [23].

The second group consists of those contributions focusing to reduce the execution time of the MT process. In this field, there are different proposals based on shared-memory, which can be divided in two different categories: Single Instruction Multiple Data systems [5, 9] and Multiple Instruction Multiple Data systems [7, 21]. The main issue of this kind of systems is the lack of scalability in the number of processors and in the memory system.

Moreover, there exist multiple contributions based on distributed memory [3, 20]. It is worth to mention the contribution of Mateo and Usaola. They presented a dynamic distribution algorithm, called PEDRO (Parallel Execution with Dynamic Ranking and Ordering), that uses Factoring Self-Schedulling ideas [19]. Also, they introduce *Bacterio*^{P}, a parallel extension of the MT tool *Bacterio* [18], that uses *Java-RMI* [6] in order to communicate processes through the network. The results obtained in this proposal are better than those obtained in previous works. However, although the performance and the scalability achieved in this contribution is better than those obtained in shared-memory approaches, the used communication mechanism acts as a system bottleneck and, consequently, it limits the overall system performance [17].

In our previous work we proposed `EMINENT` [2] , a dynamic distributed algorithm focused on HPC systems and designed to reduce the high computational cost of MT. In `EMINENT` we use MPI to interchange information between processes, which alleviates the previously described bottleneck issue [17]. However, we consider that the distribution of the workload can be improved in order to increase both the resource usage efficiency and the level of parallelism.

3 Description of `OUTRIDER`

In this paper we propose an optimization that contributes to bridging the gap between one of the main limitations of MT, its high computational cost, and the main advantages provided by HPC systems, parallel infrastructures to speed-up the execution of computational applications. In this section we present in detail our proposal, called `OUTRIDER`, which consists of 4 different strategies to improve the overall performance of the MT process.

3.1 Parallelizing the execution of the TS over the original application

Usually, the sequential execution of a TS over the original program is an issue that hampers the scalability of the MT process [19]. This phase of the testing process becomes specially relevant in those cases where the application under test requires a long execution time and when the TS consists of a large number of test cases. Consequently, the scalability of the system is compromised due to the lack of parallelism, which is generally reflected in a low system performance.

In order to alleviate this issue, we propose to exploit the resources of the system by executing the TS over the original program in parallel. Basically, this strategy consists in distributing the

execution of each test over the original program among different processes, which are executed in the available CPU cores of the system.

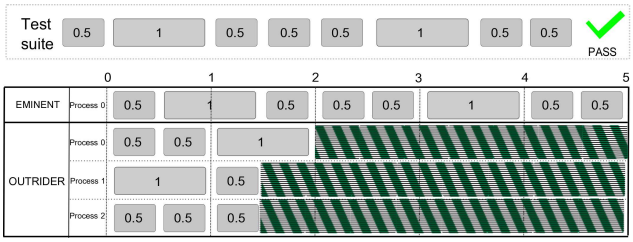


Figure 1: Execution of a TS over the original program using `EMINENT` and `OUTRIDER`

Figure 1 illustrates the comparison between the execution of a TS over the original program using `EMINENT` and `OUTRIDER`. The schema at the top of the figure shows the sequential execution of the TS, where each rectangle represents a test case and the number inside it shows the required slots of time to be executed. This TS consists of 8 test cases and has a total duration of 5 slots of time. While the TS is sequentially executed in one processor using `EMINENT`, `OUTRIDER` parallelizes the execution of the TS using 3 different processes. Also, this example shows that the distribution of the TS improves the overall performance, obtaining a speed-up of 2.5.

3.2 Sorting the TS

In MT, test cases are executed over a mutant until one of these situations occurs: the mutant is killed or the TS is completely executed and the mutant is kept alive. It is therefore desired that the first test case to be executed kills the mutant, it being the ideal situation. Unfortunately, we cannot determine those test cases that kill the mutant before executing them. However, we can use relevant information gathered from the execution of the TS over the original program, like the execution time of each test case.

This strategy uses this information to specify the execution order of the test cases. Thus, test cases are sorted by using its execution time as sorting criteria. As a result, the fastest test case is processed in the first place, while the slowest test case is executed last. The idea is to minimize the required time to kill a mutant. Although sorting the TS has a computational cost, we suppose that applying this strategy would reduce the overall execution time.

$\mathbf{TC}_{EMI}$	$\mathbf{TC}_{OUT}$	$\mathbf{ExecTime}_{EMI}$	$\mathbf{ExecTime}_{OUT}$	$\mathbf{Acc}_{EMI}$	$\mathbf{Acc}_{OUT}$	Improv.
1	2	1175	139	1175	139	-848
2	5	139	211	1314	350	1175
3	3	498	498	1812	848	964
4	1	1471	1175	3283	2023	-211
5	4	211	1471	3494	3494	3144

Table 1: Execution time, in seconds, of 5 test cases over a mutant using `EMINENT` and `OUTRIDER`

In order to illustrate the concepts of this strategy, we present a running example. Table 1 shows the execution of a TS consisting of 5 test cases over a mutant. The first two columns, TC_{EMI} and TC_{OUT}, refer to the order of execution for each test case using `EMINENT` and

OUTRIDER, respectively. The next two columns, ExecTime$_{EMI}$ and ExecTime$_{OUT}$, represent the execution time for each single test case. These are followed by the two columns that represent the accumulative time required to execute the test cases using EMINENT and OUTRIDER. The last column shows the improvement obtained by comparing OUTRIDER and EMINENT, where positive values indicate that OUTRIDER executes faster than EMINENT and negative values show otherwise. This improvement is calculated by taking into account that there is a test case that kills the mutant, which is indicated in the first column. In this example, OUTRIDER obtains better results than EMINENT when the test case 2, 3 or 5 kills the mutant, obtaining an improvement in the total execution time of 1175, 964 and 3144 seconds, respectively. On the contrary, EMINENT executes faster than OUTRIDER when test case 1 or 4 kills the mutant.

3.3 Enhancing the test case distribution strategy

In order to increase both the level of parallelism and the resource usage efficiency, we propose a strategy that improves the workload distribution presented in EMINENT, which additionally considers the number of remaining mutants to be completely executed, the number of processes involved in the testing process and the number of processes that are executing each mutant. The idea is to improve the resource usage efficiency by maximizing the number of different mutants being executed in parallel. Thus, when the number of remaining mutants to be executed is greater or equal than the number of available processes, each single process executes a different mutant. On the contrary, the remaining mutants to be completely processed are proportionally distributed among the available processes.

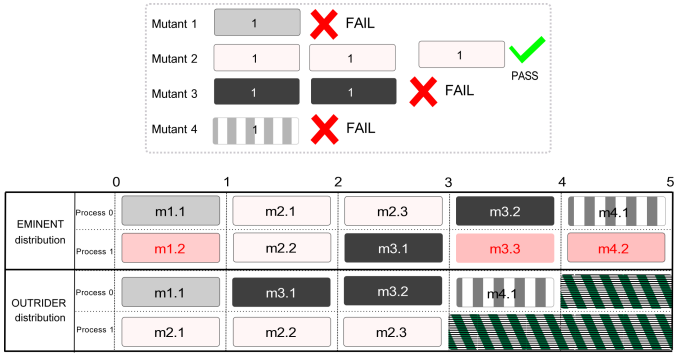


Figure 2: Workload distribution using EMINENT and OUTRIDER

Figure 2 illustrates a running example to compare two different distribution strategies. In this example a TS consisting of 3 test cases is executed over 4 mutants using 2 processes, each one having a dedicated CPU core. The schema at the top of Figure 2 shows the execution of the TS over each mutant, where test case 1 kills mutant 1 and 4, test case 2 kills mutant 3, and mutant 2 remains alive after the execution of the TS.

The schema at the bottom of Figure 2 shows two different strategies to distribute the workload in the MT process, that is, the workload distribution presented in EMINENT and the workload distribution used in OUTRIDER. For the sake of clarity, we denote by mX.Y the

execution of the test case Y over the mutant X. In this scenario, executions m1.1, m3.2 and m4.1 kill the processed mutant.

The distribution strategy used in `EMINENT` shows that the execution of some test cases is useless. For instance, m1.1 and m1.2 are executed in parallel. Although the former execution kills mutant 1, process 1 is wasting computational resources by executing m1.2, which is not necessary to kill the mutant. Since the strategy used in `OUTRIDER` maximize the number of different mutants executed in parallel, this situation is avoided in the major part of scenarios. In this example, `OUTRIDER` obtains an improvement of 20% in the total execution time.

3.4 Categorizing equivalent mutants using TCE

In MT, a mutant is considered equivalent when none of the test cases are able to kill it. The equivalence problem is one of the principal obstacles of the practical use of MT. Although it is well known that finding the equivalence between two programs is a non-decidable problem [14], there exist several heuristics that aid to find some pattern to identify this kind of mutants. In this case, due to its simplicity and its computational efficiency, we have selected the trivial compiler equivalence technique (in short, TCE), to detect both equivalent and cloned mutants [15]. This technique uses compiler optimizations in order to detect some patterns that aids to identify the equivalence between programs using a black-box scheme.

These concepts are used in our proposed strategy to detect two kinds of mutants. On the one hand, those mutants that are equivalents to the original program, which are known are *equivalents*. On the other hand, those mutants that differ from the original program but are identical to other mutants, are called *cloned* mutants.

We apply this technique after the compilation phase, where both equivalent and cloned mutants are detected. Those mutants identified as equivalents are discarded and none of them are executed. On the contrary, cloned mutants are grouped in domains, where a single mutant is known as *representative* of the domain.

During the testing phase, only those mutants that do not belong to a domain are executed, which are handled as usual. Next, for each domain, only representative mutants are processed. Once the execution of a representative mutant ends, if the mutant is killed, only the killer test is applied to the rest of the mutants of the domain, which substantially reduce the number of test case executions. On the contrary, if the representative mutant is kept alive, the rest of the mutants of the domain are managed as usual.

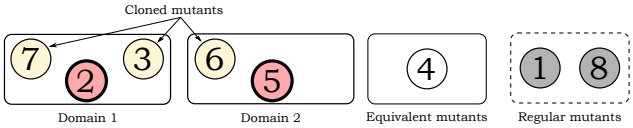


Figure 3: Categorization of cloned and equivalent mutants

In order to show the applicability of this strategy we present a running example. Figure 3 shows the execution of a MT process consisting of 8 mutants. We applied our strategy to categorize these mutants. As a result, we obtain two different domains, where Domain 1 consists of the mutant 2, 3 and 7, and Domain 2 consists of the mutant 5 and 6. Mutants with a bold border are the representatives of its domain, that is, mutant 2 is the representative mutant for Domain 1 and mutant 5 is the one for Domain 2. Mutant 4 has been detected as equivalent mutant, while mutant 1 and 8 are categorized as regular mutants.

Mutant ID	Exec.Time	Time_{EMI}	Killer test time	Time_{OUT}
1	432	432	-	432
2	245	677	-	677
3	245	922	46	723
4	456	1378	-	723
5	532	1910	-	1255
6	532	2442	164	1419
7	245	2687	46	1465
8	591	**3278**	-	**2056**

Table 2: Execution of 8 mutants using `EMINENT` and `OUTRIDER` with TCE

Table 2 shows the execution time of the MT process presented in the running example. The first two columns, *Mutant ID* and *Exec.Time*, represent the mutant ID and its execution time, respectively, $Time_{EMI}$ refers to the accumulated time when the testing process is executed using `EMINENT`. The next column refers to the execution time of the test that kills the mutant, which is calculated from the representative mutant of each domain. Finally, $Time_{OUT}$ refers to the accumulated time when the testing process is executed using `OUTRIDER`.

These results show that `OUTRIDER` executes 37% faster than `EMINENT`, that is, while `EMINENT` requires 3278 seconds to completely execute the testing process, `OUTRIDER` requires 2056 seconds. This improvement of performance is obtained because `OUTRIDER` executes less test cases than `EMINENT`. In this case, mutant 3, 6 and 7 are not completely executed because only the test case that kills them is executed instead.

4 Experiments

This section presents a thorough experimental study to analyze the scalability and performance of `OUTRIDER`. The mutant set used in these experiments has been created using the mutation framework Milu [7]. We use two different applications in the MT process. First, an image filtering application consisting of 3 algorithms to filter BMP images. Initially, all the images are located in a remote repository, which has a total size of 2,5 GB. In order to check this application, a TS consisting of 3200 test cases are executed over 250 mutants. The second application performs the multiplication of two large matrices. In this case, a TS consisting of 2000 test cases are executed over 100 mutants.

These experiments have been performed in a cluster that consists of 8 nodes interconnected through a Gigabit Ethernet network. Each node contains a Dual-Core Intel(R) Core(R) i5-3470 CPU at 3.4 Ghz with hyper-threading, 8 GB of RAM and 500GB HDD.

4.1 Performance evaluation of each single strategy in `OUTRIDER`

In this section, each proposed strategy in `OUTRIDER` is individually analyzed. For the sake of simplicity, we use the following notation: `S1` refers to the strategy that parallelizes the TS execution over the original program (see Section 3.1), `S2` refers to the strategy that sorts the TS (see Section 3.2), `S3` is the strategy that enhances the test case distribution (see Section 3.3) and `S4` is the strategy that categorizes both cloned and equivalent mutants (See Section 3.4).

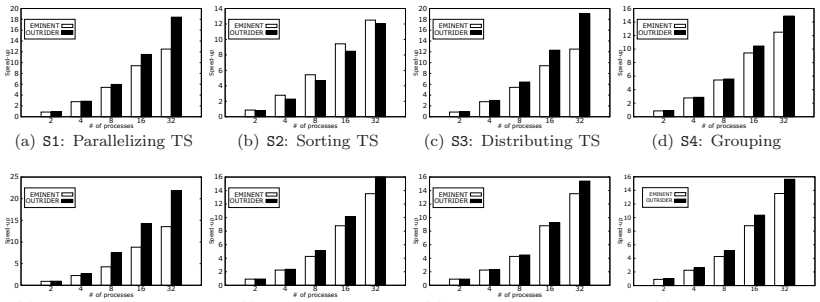
Figure 4 shows the overall speed-up in `EMINENT` and `OUTRIDER`, with respect to a sequential execution, using 2, 4, 8, 16 and 32 processes. Each process is always executed in a dedicated CPU core. In these charts, X-axis represents the number of processes and Y-axis represents the

speed-up. The first row of charts in Figure 4 shows the results obtained using the image filtering application, while the second row refers to the results obtained for checking the CPU-intensive application. Each chart analyzes a different strategy of OUTRIDER, that is, charts 4(a) and 4(e) analyze S1, charts 4(b) and 4(f) evaluate S2, charts 4(c) and 4(g) analyze S3 and charts 4(d) and 4(h) evaluate S4.

In general terms, OUTRIDER outperforms EMINENT in the major part of the evaluated scenarios. There is only one scenario where EMINENT executes faster than OUTRIDER, that is, testing the filtering application using the strategy S2 in OUTRIDER (see chart 4(b)). In this case, the major part of the mutants are killed by the first test cases of the TS without applying S2 and consequently, EMINENT requires less time to kill the mutants than OUTRIDER using a sorted TS.

The strategy that provides the best results in OUTRIDER is S1, reaching an improvement, with respect to EMINENT, of 40% in the overall execution time. This result is obtained using 32 process for testing the CPU-intensive application (see chart 4(e)). Strategy S3 also provides valuable results, reaching in some scenarios an improvement between 10% - 38% in the overall performance. However, strategy S4 provides slightly better results than EMINENT, it being the best case scenario the testing process of the image filtering application, where 2 equivalent mutants and 19 cloned mutants, divided in 13 domains, have been detected, obtaining a reduction of 20% in the total execution time.

In conclusion, OUTRIDER provides better resource usage efficiency than EMINENT. These experiments show that strategies S1 and S3 clearly provides a significant improvement in the overall system performance. Moreover, in terms of scalability, that is, the performance obtained when the computational resources are increased, these strategies are better than S2 and S4. The main reason of this behaviour is two-fold: First, strategies S1 and S3 are less dependent of both the TS and the mutant set, which mainly focus on exploiting the resources of the system. Second, strategies S2 and S4 strongly depend of the TS and the mutant set, respectively. Only in those cases where the killer test is not located in the first positions of the TS and the mutant set contains several cloned and equivalent mutants, OUTRIDER using S2 and S4 executes faster than EMINENT.



(a) S1: Parallelizing TS (b) S2: Sorting TS (c) S3: Distributing TS (d) S4: Grouping

(e) S1: Parallelizing TS (f) S2: Sorting TS (g) S3: Distributing TS (h) S4: Grouping

Figure 4: Performance of the testing process using EMINENT and OUTRIDER with a single strategy

4.2 Performance evaluation using different strategies in OUTRIDER

In this section we analyze the performance of OUTRIDER when different strategies are used. For the sake of clarity, we only show those configurations that obtain the most representative results, which are depicted in Table 3.

	Configuration							
Strategy	C_{12}	C_{13}	C_{23}	C_{123}	C_{34}	C_{134}	C_{234}	C_{1234}
S1: Parallelizing TS	✓	✓		✓		✓		✓
S2: Sorting TS	✓		✓	✓			✓	✓
S3: Distribution		✓	✓	✓	✓	✓	✓	✓
S4: Grouping					✓	✓	✓	✓

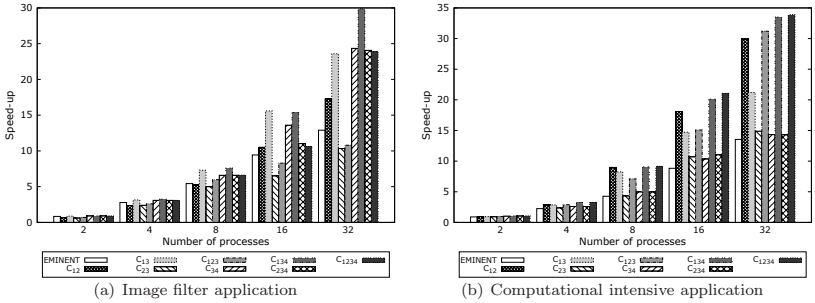
Table 3: Configuration of the strategies used in OUTRIDER

Figure 5 shows the results of executing the testing process with EMINENT and OUTRIDER using different configurations. In these charts, X-axis shows the number of processes used and Y-axis shows the obtained speed-up with respect to a sequential execution.

Figure 5(a) presents the results of the testing process using the image filtering application. In general, OUTRIDER achieves better performance than EMINENT. Both approaches provide similar performance when 2 and 4 processes are used. However, when the number of computational resources increases, the difference of performance between EMINENT and OUTRIDER increases as well. For instance, when 32 processes are used, OUTRIDER with $C_{13}, C_{34}, C_{134}, C_{1234}$ obtains a reduction in the total execution time, with respect to EMINENT, of 50%, 50.5%, 60% and 50%, respectively. Configuration C_{134} is particularly relevant. In this case, OUTRIDER achieves a speed-up of 2.3 with respect to EMINENT. This improvement in the overall testing performance is mainly reached because of the S4 strategy, which accelerates the MT process by avoiding the complete execution of some mutants (see Section 3.4). However, these configurations using the strategy S2 do not guarantee an improvement in the total execution time. For instance, OUTRIDER using C_{23} executes 19% slower than EMINENT.

Figure 5(b) shows the results of the testing process using the CPU-intensive application. Similarly to the previous experiments, we obtain a similar tendency in the system scalability. That is, using few computational resources provides almost the same performance for both approaches. However, increasing the number of computational resources provides a proportional improvement in the overall system performance. In these experiments, all the configurations used in OUTRIDER provide a better performance than EMINENT. In particular, $C_{12}, C_{123}, C_{134}, C_{1234}$ are especially relevant because OUTRIDER using these configurations executes 62%, 67%, 70% and 71% faster than EMINENT, respectively.

In general, the overall system performance is increased when combining different strategies. Configuration C_{134} achieves a significantly better speed-up than EMINENT for checking the image filtering application, especially when 32 processors are used. However, configuration C_{1234} only achieves slightly better results than C_{134} when 8 and 16 processors are used to check the CPU-intensive application. It is important to remark that the best results are obtained when different strategies are combined using OUTRIDER, especially strategies S1 and S4. In conclusion, OUTRIDER provides a better resource usage efficiency than EMINENT, which is reflected in the scalability obtained, reaching in some cases a speed-up higher than the number of CPU cores used.



(a) Image filter application (b) Computational intensive application

Figure 5: Performance evaluation of `EMINENT` and `OUTRIDER` using different strategies

5 Conclusions

In this paper we have presented `OUTRIDER`, an HPC-based optimization for the MT process in HPC systems. This optimization consists of 4 strategies aimed to improve the resource usage efficiency, which uses `EMINENT` as basis. Also, an experimental phase has been carried out to evaluates the effectiveness and scalability of `OUTRIDER`.

The experimental study carried out in this work shows that `OUTRIDER` outperforms previous proposals to improve performance of the MT process. In general, `OUTRIDER` provides the best results when different strategies are combined, specially `S1`, `S3` and `S4`, obtaining in some scenarios an improvement of 70% in the overall performance with respect to `EMINENT`. On the contrary, the results obtained when `S2` is used shows that an improvement in the overall performance is not guaranteed. For instance, there are scenarios where `OUTRIDER` executes 66% faster than `EMINENT` (see C_{12} and C_{123} in Figure 5(b)) and there is other scenarios where `OUTRIDER` executes 20% slower than `EMINENT` (see C_{23} in Figure 5(a)).

As future work, we will evaluate the possibility to include some mechanisms for automatically selecting these strategies to be applied in a given MT environment.

References

[1] Sergey Bastrakov, Iosif Meyerov, Victor P. Gergel, Arkady Gonoskov, Anton V. Gorshkov, Evgeny Efimenko, M. Ivanchenko, Mikhail Yu. Kirillin, A. Malova, G. Osipov, V. Petrov, Igor Surmin, and A. Vildemanov. High performance computing in biomedical applications. In *International Conference on Computational Science*, pages 10–19, 2013.

[2] P.C. Cañizares, M.G. Merayo, and A. Núñez. Eminent: Embarrassingly parallel mutation testing. In *International Conference on Computational Science*, volume 80, pages 63–73. Elsevier, 2016.

[3] B. Choi and A.P. Mathur. High-performance mutation testing. *Journal of Systems and Software*, 20(2):135–152, 1993.

[4] L. Deng, A.J. Offutt, P. Ammann, and N. Mirzaei. Mutation operators for testing android apps. *Information and Software Technology*, 2016.

[5] R. Gopinath, C. Jensen, and A. Groce. Topsy-turvy: a smarter and faster parallelization of mutation analysis. In *International Conference on Software Engineering Companion*, pages 740–743. ACM, 2016.

[6] W. Grosso. *Java RMI*. O'Reilly & Associates, Inc., 1st edition, 2001.

[7] Y. Jia and M. Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language. In *Practice and Research Techniques*, pages 94–98. IEEE, 2008.

[8] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, and N. Malevris. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In *International Working Conference on Source Code Analysis and Manipulation*, pages 147–156. IEEE, 2016.

[9] E.W. Krauser, A.P. Mathur, and Vernon J. Rego. High performance software testing on simd machines. *IEEE Transactions on Software Engineering*, 17(5):403–423, 1991.

[10] F. Li, J. Yu, Z. Cao, J. Zhang, M. Chen, and X. Li. Experimental demonstration of four-channel wdm 560 gbit/s 128qam-dmt using im/dd for 2-km optical interconnect. *Journal of Lightwave Technology*, 2016.

[11] Y. Ma and S. Kim. Mutation testing cost reduction by clustering overlapped mutants. *Journal of Systems and Software*, 115:18–30, 2016.

[12] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst D. Simon. Top500 supercomputer sites, 2016. `http://www.top500.org`.

[13] R.M. Hierons M.G., Merayo, and M. Núñez. Controllability through nondeterminism in distributed testing. In *International Conference on Testing Software and Systems*, pages 89–105, 2016.

[14] A.J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software testing, verification and reliability*, 7(3):165–192, 1997.

[15] M. Papadakis, Y. Jia, M. Harman, and Y. Le-Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *International Conference on Software Engineering*, volume 1, pages 936–946. IEEE, 2015.

[16] D.R. Penas, P. González, J.A. Egea, J. R. Banga, and R. Doallo. Parallel metaheuristics in computational biology: An asynchronous cooperative enhanced scatter search method. In *International Conference on Computational Science*, pages 630–639, 2015.

[17] K. Qureshi and H. Rashid. A performance evaluation of rpc, java rmi, mpi and pvm. *Malaysian Journal of Computer Science*, 18(2):38–44, 2005.

[18] P. Reales and M. Polo. Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases. In *International Conference on Software Maintenance*, pages 646–649. IEEE, 2012.

[19] P. Reales and M. Polo. Parallel mutation testing. *Journal of Software Testing, Verification and Reliability*, 23(4):315–350, 2013.

[20] I. Saleh and K. Nagi. Hadoopmutator: A cloud-based mutation testing framework. In *Software Reuse for Dynamic Systems in the Cloud and Beyond*, pages 172–187. Springer, 2014.

[21] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for java. In *International Conference of Software Engineering Conference*, pages 297–298. ACM, 2009.

[22] E. Wong. *On mutation and data flow*. PhD thesis, Purdue University, 1993.

[23] F. Wu, M. Harman, Y. Jia, and J. Krinke. Homi: Searching higher order mutants for software improvement. In *International Symposium on Search Based Software Engineering*, pages 18–33. Springer, 2016.

7.6 MAGICIAN: Model-based design for optimizing the configuration of data-centers

7.6	
Publication data	
Authors:	Pablo C. Cañizares, Alberto Núñez and Juan de Lara
Title:	MAGICIAN: Model-based design for optimizing the configuration of data-centres
Publication:	29^{th} International Conference on Software Engineering and Knowledge Engineering
Pub. Type:	Conference
Year:	2017
DOI/URL:	https://doi.org/10.18293/SEKE2017-108
Pages:	6
CORE Ranking:	B
Contribution	
Summary:	This paper describes a model-based approach to design data-centres. For this purpose, a meta-model that describes the structure of data-center models has been created. Then, a set of expert rules can be used to detect sub-optimal configurations, and (in some cases) correct the design. Data-centre models can be simulated, to assess their performance and scalability, for which we use a code generator into the SIMCAN tool. The approach has been implemented as an Eclipse plugin, and illustrate the usefulness of some expert rules by showing the efficiency and scalability gains of the optimized model with respect to the original one.
Technique:	Model Driven Engineering
Secondary techniques:	Simulation

MAGICIAN: Model-based design for optimizing the configuration of data-centers

Pablo C. Cañizares ★ Alberto Núñez ★ Juan de Lara ☆

★ Universidad Complutense de Madrid
Spain

☆ Universidad Autónoma de Madrd
Spain

Abstract

Designing data-centers that provide an acceptable cost-performance ratio is challenging. Generally, a wide spectrum of components must be previously analyzed, such as the kind of applications to be executed in the data-center, computing/storage requirements and the network topology, among others. Since each one of these components has a direct impact on the overall system performance, the design process is complex and difficult, which usually requires the intervention of an expert.

We propose a model-based approach to design data-centers. For this purpose, we have created a meta-model that describes the structure of data-center models. Then, a set of expert rules can be used to detect sub-optimal configurations, and (in some cases) correct the design. Data-center models can be simulated, to assess their performance and scalability, for which we use a code generator into the SIMCAN tool. We have implemented our approach as an Eclipse plugin, and illustrate the usefulness of some expert rules by showing the efficiency and scalability gains of the optimized model with respect to the original one.

1 Introduction

During the last decade, most efforts in scientific applications were focused in obtaining the best possible performance, exploiting the system resource usage both in supercomputers and commodity clusters.

Due to the high number of inter-related parameters that have a direct impact on the overall performance, building a system that provides the maximum performance for a given application is a very complex task. Designing and configuring a data-center that properly exploits the system resource usage may be a feasible task for an expert [1]. However, when the data-center is designed by a non-expert, it may provide an overall performance far from the expected one. Generally, a misconfiguration of the system architecture or a wrong choice of hardware resources, may lead to obtaining a poor performance.

Usually, the first step before deploying the data-center in a production environment consists in modelling and simulating its underlying architecture. Thus, the obtained results from the simulation are used to polish and improve the initial design. Unfortunately, the number of possible configurations is extremely large, making unpractical to model and simulate all of them.

In this paper we propose `MAGICIAN`, an approach that aids designers to optimize the configuration of data-centers. The main objective of `MAGICIAN` is to identify possible inconsistencies in the initial design of the data-center and to suggest feasible corrections. Thus, a reduced number of data-centers designs are generated, which can be simulated to analyze which one provides the best results.

The approach is based on model-driven engineering (MDE) [2]. We propose a meta-model for data-centers, so that data-center configurations are expressed as instances of such meta-model. We provide a library of expert knowledge rules to detect misconfigurations and suggest improvements on the design. Finally, we support the simulation of the data-center configuration to assess properties like scalability, and detect possible bottlenecks. The simulation is performed by generating code for the SIMCAN tool [3].

The rest of this paper is organized as follows. Section 2 provides an overview of the proposed approach. Section 3 describes in detail the principal components of `MAGICIAN`. Next, Section 4 presents performance experiments that show the usefulness of our approach. Section 5 presents related work, and Section 6 ends with the conclusions and future work.

2 Overview

In this section, we describe the `MAGICIAN` approach for optimizing the configuration of data-centers. The overall scheme is shown in Figure 1.

Our architecture represents data-center designs as models conformant to a meta-model, and includes two optimization loops. The first one is based on expert rules, which encode knowledge on typical good configurations. The second is based on simulation, with which one can analyse aspects like efficiency or scalability of the data-center models.

The structure of data-centers, and the integrity constraints that valid data-center models should obey, have been captured through a meta-model. The designer will be able to create models that *conform* to such meta-model. That is, these models use the types and relations defined in

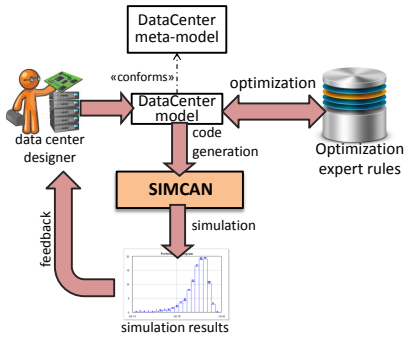


Figure 1. Working scheme of our approach

the meta-model, and satisfy the integrity constraints. This meta-model will be explained in Section 3.1.

We have created a library of expert rules, containing optimization patterns and idioms, typically followed by good data-center designs. These rules detect parts of the model that are amenable to optimization, signalling potential deficiencies in the model. Moreover, some of these rules contain quick fixes, which modify the design to improve some suboptimal aspect of the model. Technically, these rules have been implemented using the Epsilon Validation Language (EVL) [4]. They will be detailed in Section 3.2.

We also enabled the evaluation of the data-center model through simulation. This way, we have built a code generator that produces code to be executed by the SIMCAN simulation tool [3]. The results of the simulation can be used by the designer to find problems in the design, such as bottlenecks, to further improve it. The approach to code generation, and details on how the simulation is performed are given in Section 3.3.

3 Model-based simulation of data-centers

In this section we explain the three main building blocks of our approach: the data-center meta-model (see Section 3.1), the expert rules and quick fixes (see Section 3.2), and the code generation and simulation (see Section 3.3).

3.1 The data-center meta-model

Figure 2 shows a simplified version of the data-center meta-model. The *DataCenter* meta-class is the root class, which contains the main elements of a real data-center, such as those relating with both computational and networking aspects.

The computing elements are divided in two types. The first type corresponds to the *Node* meta-class, which represents a single computational node. The first 3 attributes define the CPU processor, where *CPU_Sockets*, *CPU_Cores* and *CPU_Speed* represent the number of CPUs, the number of cores of each CPU and the CPU speed (measured in MIPS), respectively. The next 3 attributes are related with memory features, where *RAM_slots*, *RAM_Size* and *RAM_Frequency* represent the number of memory modules, the total size of each module (measured in GBytes) and the frequency of the memory (measured in Mhz), respectively. The next 4 attributes refer to storage aspects, where *Disk_Slots*, *Disk_Size* *Disk_RBandwidth* and *Disk_WBandwidth* are the number of disks, the size of each disk (measured in GBytes) and the read and write bandwidth of the storage system (measured in Gbps), respectively. The last attribute, *isComputingNode*, denotes if a given node is a computing node or a storage node. The second type of computing elements corresponds to the *Rack* meta-class. A rack represents a structure that contains multiple computing elements. In this case, the rack consists of a set of *Boards*, where each board contains a number of nodes that is determined by the attribute *Nodes_per_board*.

The network is defined by 2 elements. The *Network* meta-class represents the communication network of the data-center, where *Bandwidth*, *Latency*, and *ErrorRatio* define the data transfer rate (measured in Gbps), the latency (measured in μs) and the error ratio of the network, respectively. The *Switch* meta-class represents a resource used to communicate the different computing elements of the data-center through the communication network, where *MTU* and *NumPorts* are the maximum transmission unit and the number of ports of the switch, respectively.

Finally, the *Repository* meta-class represents the data-center repository, which provides a wide collection of networking and computational components to model, with a high level of detail, a complete data-center.

Figure 3 shows an example of a data-center model which conforms the proposed meta-model. This model is inspired by a real IBM data-center configuration, which consists of 1 *IBM Flex* rack and 1 *IBM v7000 Storage* rack. These racks are interconnected using a 40/10 Gigabit Ethernet communication network and a *SAN42B-R extension switch*. The *IBM Flex* rack consists of 6 *Flex System Enterprise Chassis* boards, where each board contains 14 *IBM Flex System p460* computing nodes. These boards consist of 4 CPUs with 8 cores, reaching a speed of 317.900 MIPS. The memory system consists of 32 slots, which contain 64 GB of RAM. Finally, the storage consists of two disks of 2 TBytes. The *IBM Storage* rack consists of 6 *Flex System Chassis Storage* boards, where each board contains 14 *IBM v7000* storage nodes. Each node consists of 2 CPUs with 4 cores, reaching an speed of 200.000 MIPS, 16 GB of RAM and 16 hard disks with a total storage of 10 TBytes.

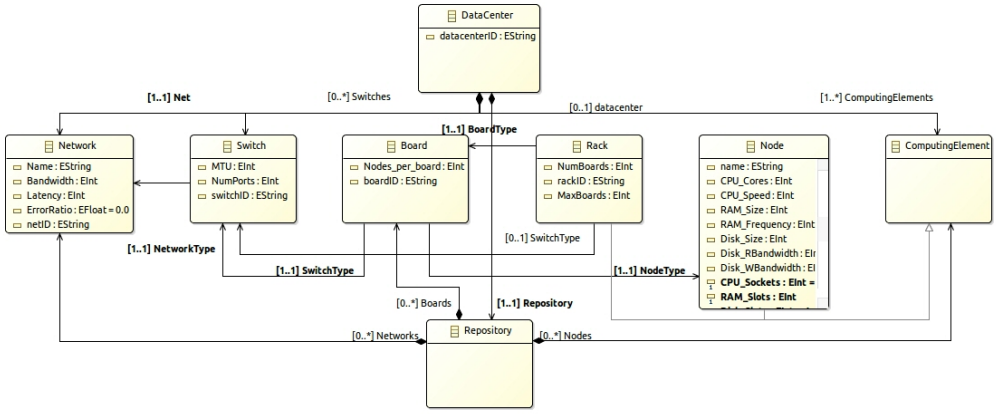


Figure 2. The data-center meta-model (excerpt)

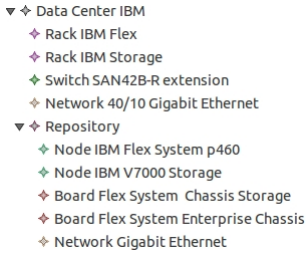


Figure 3. Example of a data-center model

3.2 Encoding expert rules and heuristic quick fixes

In order to support the user during the data-center design process, we have included a library of optimization rules based on data-center experts knowledge. These expert rules aid the user to solve possible design issues, which in most cases, hamper the overall data-center performance. However, expert rules must be designed and provided by an expert user, who must decide whether these are suitable to cover the requirements of the systems under study. The library consists of several rules focused on analysing different features of the data-center components, such as CPU processors, the memory system, storage and connectivity, among others. The main goal of these rules is to find inconsistencies in data-center models and to provide relevant information to fix them. For the sake of simplicity, in this paper we have described a sample of three rules from the complete library.

Listing 1 shows the expert rule *CoresVsStorageNodesRatio* encoded in EVL. This rule analyses the ratio between the number of storage nodes and the number of CPUs of a data-center. The main objective of this rule is to avoid system bottlenecks caused by a reduced number of storage nodes. In this case, if the available storage nodes are not able to provide the required performance, a message to modify the current design is shown. As can be seen, the rule is applied on the context of DataCenter objects (line 1 of the listing). It is made of a check section (lines 5–11), which evaluates a certain condition on the model, and a message part, which is presented to the designer if the check part returns true.

```
1  context DataCenter
2  {
3    critique CoresVsStorageNodesRatio
4    {
5      check{
6        var storageNodes: Integer;
7        var totalCores: Integer;
8        storageNodes = self.calculateStorageNodes();
9        totalCores = self.calculateTotalCores();
10       return storageNodes*40 >= totalCores;
11     }
12     message: 'The number of storage nodes must be increased, there
             exist a high number of cores in comparison with the number of
             storage node which can act as bottleneck'
13   }
14 }
```

Listing 1. Data-center topology optimization rule encoded in EVL

Listing 2 shows two expert rules based on the analysis of two network features, bandwidth and latency. In this case, these rules check that these features range in a determined interval. If some of these features is out of the range, the system provides a quick-fix method to alleviate the issue.

Quick fixes are specified in the fix section of the rules (lines 7–9 and 16–19). Figure 4 shows how such quick-fix is presented to the user.

```
1  context Network
2  {
3    critique NetBandwidth
4    {
5      check: self.Bandwidth>=10 and self.Bandwidth <=100
6      message: 'Network '+ self.Name + ': Bandwidth is usually ranged in
             [10−100]. Some of the most used configuration is 40'
7      fix {
8        title : "Set Bandwidth" + self.Name + " Bandwidth to 40"
9        do { self.Bandwidth = 40; }
10     }
11   }
12   critique NetLatency
13   {
14     check: self.Latency>=20 and self.Latency <=2000
15     message: 'Network '+ self.Name + ': Latency is usually ranged in
             [20−2000]. Some of the most used configuration is 200'
16     fix {
17       title : "Set Latency" + self.Name + " Latency to 40"
18       do { self.Latency = 200;}
19     }
20   }
21 }
```

Listing 2. Network optimization rules encoded in EVL

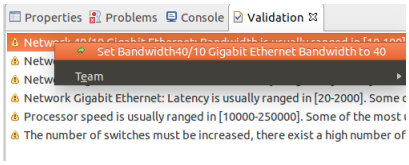


Figure 4. Example of network quick-fix

3.3 Code generation and Simulation

Once the data-center has been modelled, it can be simulated, to analyse its efficiency, in terms of scalability and performance. In this paper, we use the SIMCAN simulation platform to represent and simulate the behaviour of data-centers [3]. We have created a code generator that transforms the designed model into the required configuration files to perform the simulation.

Listing 3 shows an extract of the generated data-center topology, written in the NED language. The first line represents the name of the data-center, the next 3 lines refers to the different resources that compose the environment, such as switch, storage and computing nodes, respectively. Finally, the lines 6-8 show how the storage and computing elements are connected through the communication network, using the switch component.

```
1  network IBM{
2    switch_0:EtherSwitch;
3    rCmp1_IBM_Flex_Rack:Rack;
4    rSto0_StorageRack:Rack;
5
6    for i=0..5 {
7      rCmp1_IBM_Flex_Rack.ethg++ <−> Eth10M_channel <−>
           switch_0.ethg++;
8      rSto0_StorageRack.ethg++ <−> Eth10M_channel <−> switch_0.
           ethg++;
9    }
10 }
```

Listing 3. Example of data-center topology in SIMCAN written in NED language (excerpt)

Listing 4 shows an excerpt of a generated data-center configuration file. This portion of the configuration file configures the computing rack illustrated in Figure 3. It is important to remark that the symbol ∗ refers to a wildcard that represents all the elements in the referenced structure. For example, the lines 3 and 11 refer to the configuration of the network for all the boards in the rack.

```
1  IBM.rCmp1_IBM_Flex_Rack.numBoards = 6
2  IBM.rCmp1_IBM_Flex_Rack.nodesPerBoard = 14
3  IBM.rCmp1_IBM_Flex_Rack.nodeBoard[∗].channelType = "Eth10M"
4  IBM.rCmp1_IBM_Flex_Rack.nodeBoard[∗].node[∗].cpuModule.CPUcore
       [∗].speed = 79475
5  IBM.rCmp1_IBM_Flex_Rack.nodeBoard[∗].node[∗].bsModule[∗].disk.
       readBandwidth = 650.0Mbps
6  IBM.rCmp1_IBM_Flex_Rack.nodeBoard[∗].node[∗].bsModule[∗].disk.
       writeBandwidth = 420.0Mbps
7  IBM.rCmp1_IBM_Flex_Rack.nodeBoard[∗].node[∗].osModule.memory.
       size = 2.0GiB
8
9  IBM.rSto0_StorageRack.numBoards = 1
10 IBM.rSto0_StorageRack.nodesPerBoard = 1
11 IBM.rSto0_StorageRack.nodeBoard[∗].channelType = "Eth10M"
12 IBM.rSto0_StorageRack.nodeBoard[∗].node[∗].bsModule[∗].disk.
       readBandwidth = 650.0Mbps
13 IBM.rSto0_StorageRack.nodeBoard[∗].node[∗].bsModule[∗].disk.
       writeBandwidth = 420.0Mbps
14 IBM_ScenarioA_1server.rSto0_StorageRack.nodeBoard[∗].node[∗].
       osModule.memory.size = 2.0GiB
```

Listing 4. Data-center configuration in SIMCAN

4 Evaluation

This section presents a experimental study that shows the applicability of our our proposed approach. In order to carry out these experiments, a data-center inspired by IBM Flex system has been modelled (see Figure 2). In this case, the target system contains two racks and one main switch, that is, one computing rack for processing purposes and one storage rack for managing data. The computing rack consists of 6 board nodes, where each board contains 14 p460 blades with 4 CPUs, 64 GB of RAM memory and a local disk drive of 1TB. Hence, the modelled system provides a total of 336 CPUs. The storage rack has been modelled with one blade consisting of 2 CPUs, 32GB of RAM memory

and a high performance disk drive of 2TB. Each rack uses an Ethernet 10/100 network to interconnect the blades. The main switch is connected to each rack through an Ethernet 10-Gigabit network.

This data-center has been modelled using MAGICIAN, and the alternative designs have been simulated using SIMCAN[3], using the code generator explained in Section 3.3. In order to analyze the overall system performance, a Map-Reduce application has been used [5]. This application processes a 2.5GB data-set. This data-set is divided into small data portions, called domains, which are delivered among the different processes. In these experiments, 336 processes are executed in the available CPUs of the system. It is important to remark that each process has a dedicated CPU. The size of each domain is 4MB and the size of generated data, after processing each domain, is 2MB. Each process requires 1,875,000 MIs to process a single domain.

Once the data-center has been modelled, MAGICIAN detects 2 possible inconsistencies in the data-center configuration. The first inconsistency targets the infrastructure of the data-center (the rule detecting this issue is the one in Listing 1), while the second is related to a possible misconfiguration of a single parameter (see Listing 2).

In the first case, the storage rack has been configured to use 1 blade only, that is, 1 storage server. Generally, when the proportion between the number of processes and the storage resources is not properly balanced, the storage system acts as a system bottleneck, slowing down the overall system performance. Consequently, MAGICIAN suggests to increase the number of storage servers by modelling each board in the rack with different storage blades. In order to show the usefulness of this rule, different alternative configurations, using the expert rule described in Listing 1, have been generated. In this case, these simulations have been executed using 1, 2, 4, 8, 16 and 32 storage servers. Figure 5 shows the results obtained from these simulations, where the x-axis represents the number of storage servers and the y-axis represents the speed-up with respect to the initial configuration using 1 storage server. This chart shows that the overall system performance slightly increases when the number of storage servers increases as well. However, this chart also shows that there should be another bottleneck in the system, because the increase is not high.

In the second case, MAGICIAN suggests to change the configuration of the network. Since the original model uses a 10/100 Ethernet network in each rack, it may lead to slowing down the system significantly. Similarly, in this case we have simulated the data-center using a different number of storage servers and a 10-Gigabit Ethernet for communicating the blades in the racks. Figure 6 shows the results obtained from the simulation of the alternative data-center designs. This chart shows a significant increment of performance when the number of storage servers are increased.

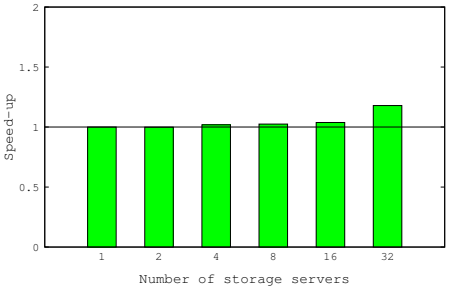


Figure 5. Proposed data-center designs by the expert-rule shown in Listing 1

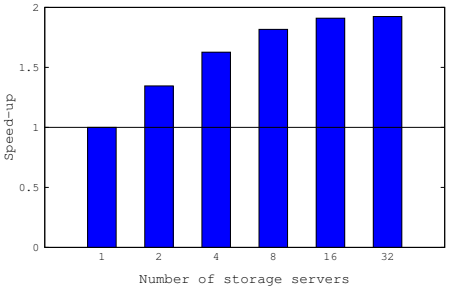


Figure 6. Proposed data-center designs by the expert-rule shown in Listing 2

As a conclusion, the initial configuration of the data-center had two drawbacks. First, using only 1 storage server limits the parallelism for accessing data in the system. In this case, MAGICIAN detects an inconsistency in the ratio between the number of CPUs and the number of storage servers. The expert rule suggests to increase the number of storage servers. In particular, this rule recommends to use 8 storage servers for this data-center configuration. Second, the network used in the racks acts as a system bottleneck. In this case, the issue is easily fixed by using the corresponding quick-fix.

5 Related work

The correct design of distributed systems is a process that requires years of expertise. In order to alleviate the inconveniences of this complex and costly task

the scientific community has performed a constant effort [6, 7]. Alshahrani and Peyravi presented a theoretical model to design and evaluate communication networks in data-centers [8]. This proposal includes an experimental analysis where the three major DCN architectures have been deployed by using simulation techniques.

In the field of modelling and simulation several contributions can be found. Son et. al presented CloudSimSDN [9], a simulation framework for software-defined cloud infrastructures. This framework incorporates a graphical interface to design the data-center topology. Meisner et. al presented [10], a simulation infrastructure for data-center systems. This approach is based on a higher level of abstraction, and uses a combination of queuing theory and stochastic modeling, which reduce the overall simulation time. Although these works allow to model several infrastructures in a fast and easy way, some of their main weakness are related with the low level of detail of the resultant infrastructures models. In order to alleviate these inconveniences, Nuñez et. al presented SIMCAN [3], a simulation platform designed to analyse and test parallel and distributed architectures and applications. In addition, a graphical user interface to help users without specific knowledge with the task of modelling new architectures is included.

More recently, Palyart et al. presented MDE4HPC [11], an model-based approach to describe and generate scientific knowledge for diverse architectures. This work presents a methodology to generate HPC applications independently from the platform by using Archi-MDE. Hence, to the best of our knowledge, there is no proposal to design data-center infrastructures that combines expert rules and simulations. Although there exist several simulation platforms, none of them includes users assistance during the modelling process. For this, our approach complements some of the existing simulation platform with expert-rules. In this case, we have selected SIMCAN due to its high level of detail and flexibility. In addition, this SIMCAN simulation platform is based on the OMNeT++, one of the most extended and adopted simulation platforms in the scientific community. Moreover, expert rules are expressed in EVL, which in its turn is based on OCL, a widely used standard for expressing model queries and constraints.

6 Conclusions and Future work

In this work we have presented `MAGICIAN`, a model-based approach for the design and analysis of data-center configurations. The metodology relies on expert rules to detect and fix suboptimal decisions, and on simulation to analyse performance and scalability of the configurations.

We have performed several experiments by modelling a real data-center using `MAGICIAN`. The proposed data-center designs, after applying the suggestion made by `MAGICIAN`, show that the existent inconsistencies in the initial design are fixed. Also, the new designs provide an overall system performance higher than the initial model.

In the future, we would like to support semi-automatic tuning configuration to reach a specific performance goal. We are also planning to identify recurring architectural patterns, which can be expressed as configurable templates.

Acknowledgements

Research partially supported by the Spanish MINECO projects DArDOS (TIN2015-65845-C3-1-R) and FLEXOR (TIN2014-52129-R), and the Comunidad de Madrid project SICOMORo-CM (S2013/ICE-3006).

References

[1] S. Tarapore, C. Smullen, and S. Gurumurthi, "Midas: An execution-driven simulator for active storage architectures," in *Workshop on Modeling, Benchmarking, and Simulation*, Beijing, 2008, pp. 1–10.

[2] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, ser. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[3] A. Nuñez, J. Fernández, R. Filgueira, F. García, and J. Carretero, "SIMCAN: A flexible, scalable and expandable simulation platform for modelling and simulating distributed architectures and applications," *Simulation Modelling Practice and Theory*, vol. 20, no. 1, pp. 12–32, 2012.

[4] S. Kolovos, R. Paige, and F. Polack, "Rigorous methods for software construction and analysis." Springer, 2009, ch. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages, pp. 204–218.

[5] A. Núñez, C. Andrés, and M. G. Merayo, "Optimizing the Trade-offs Between Cost and Performance in Scientific Computing," in *International Conference on Computational Science*, 2012, pp. 498–507.

[6] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-efficient sampling for performance prediction of configurable systems (t)," in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 342–352.

[7] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 284–294.

[8] R. Alshahrani and H. Peyravi, "Modeling and simulation of data center networks," in *Conference on Principles of Advanced Discrete Simulation*. ACM, 2014, pp. 75–82.

[9] J. Son, A. V. D., R. Calheiros, X. Ji, Y. Yoon, and R. Buyya, "Cloudsimsdn: Modeling and simulation of software-defined cloud data centers," in *International Symposium on Cluster, Cloud and Grid Computing*. IEEE Computer Society, 2015, pp. 475–484.

[10] D. Meisner, J. Wu, and R. Wenisch, "Bighouse: A simulation infrastructure for data center systems," in *Int. Symp. Performance Analysis of Systems and Software*. IEEE Comp. Soc., 2012, pp. 35–45.

[11] M. Palyart, D. Lugato, I. Ober, and J. Bruel, "MDE4HPC: an approach for using model-driven engineering in high-performance computing," in *Integrating System and Software Modeling*, vol. 7083. Springer, 2011, pp. 247–261.

7.7 FARTHEST: FormAl distRibuTed scHema to dEtect Suspicious arTefacts

7.7	
Publication data	
Authors:	Pablo C. Cañizares, Mercedes G. Merayo and Alberto Núñez
Title:	FARTHEST: FormAl distRibuTed scHema to dEtect Suspicious arTefacts
Publication:	8th Asian Conference on Intelligent Information and Database Systems
Pub. Type:	Conference
Year:	2016
DOI/URL:	https://doi.org/10.1007/978-3-662-49381-6_74
Pages:	11
CORE Ranking:	N.A.
Contribution	
Summary:	In this paper we propose a formal distributed schema, formally specified and analysed, to detect suspicious artefacts
Technique:	Formal Modelling
Secondary techniques:	Simulation, Cloud Computing

FARTHEST: FormAl distRibuTed scHema to dEtect Suspicious arTefacts

Pablo C. Cañizares( ), Mercedes G. Merayo, and Alberto Núñez

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid, Madrid, Spain
{pablocc,mlmgarci,albenune}@ucm.es

Abstract. Security breaches are a major concern by both governmental and corporative organisations. This is the case, among others, of airports and official buildings, where X-ray security scanners are deployed to detect elements representing a threat to the human life. In this paper we propose a formal distributed schema, formally specified and analysed, to detect suspicious artefacts. Our approach consists in the integration of several image detection algorithms in order to detect a wide spectrum of weapons, such as guns, knifes and bombs. Also, we present a case of study, where some performance experiments are carried out for analysing the scalability of this schema when it is deployed in current systems.

1 Introduction

After the 11/9 tragedy, security has become a national priority for governments around the world. Consequently, high security measures have been imposed in both governmental and corporative facilities to ensure the integrity of citizens. It is worth to emphasise the case of the airports, where only the U.S. government generated an annual $700 million market [10701] in the deployment of *Explosives Detection Systems* (EDS) [MR95], which are based on X-ray imaging for scanning baggage (see Fig. 1). The massive deployment of EDS has arisen the interest of the scientific community. Consequently, several detection techniques have been reported in the literature [WB12, Mer15a], like artificial neural networks [LW08], SVM [NPA08] and novel multiple-view approaches [Mer15b, US15]. All these techniques, especially the multiple-view ones, have high detection rates for processing all type of threat artefacts such as guns, weapons, bombs and knifes, which represent a beneficial contribution to protect the human life. However, these mentioned techniques are designed to be executed in a single machine, it being overexposed to risks such as computer attacks, lack of fault tolerance and replica.

Research partially supported by the Spanish MEC projects ESTuDIo and DArDOS (TIN2012-36812-C02-01 and TIN2015-65845-C3-1-R) and the Comunidad de Madrid project SICOMORo-CM (S2013/ICE-3006).

N.T. Nguyen et al. (Eds.): ACIIDS 2016, Part I, LNAI 9621, pp. 770–779, 2016.
DOI: 10.1007/978-3-662-49381-6_74

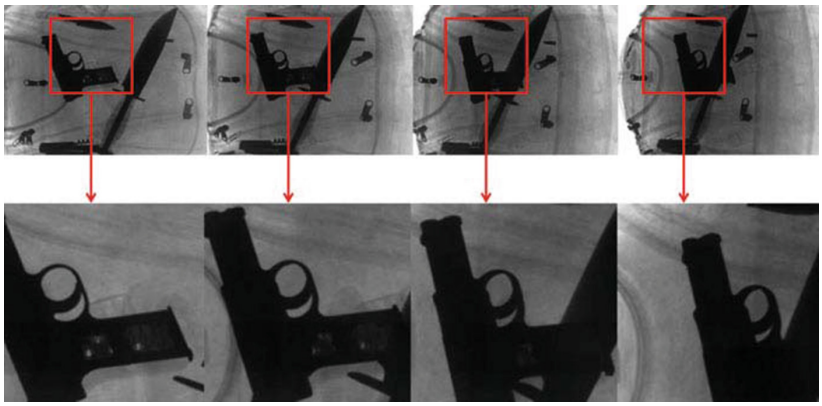


Fig. 1. X-ray image for detecting guns in baggage [Mer15b]

In order to avoid security risks, it is important to incorporate mechanisms to increase the confidence on the correctness of a system with respect to a specification. Thus, we consider that formal methods should be used to develop critical systems. Formal methods are techniques based on mathematics for modelling complex systems and to represent the specification, development, and verification of both software and hardware systems. It is important to mention that the combined use of formal methods and testing techniques [CHN15] allows us to ensure the fulfilment of a specific set of requirements and it is especially useful to detect unexpected behaviours.

In this paper we propose a distributed schema, called `FARTHEST`, to detect suspicious artefacts. We have used a formal approach to specify and analyse `FARTHEST`. In addition, we provide several specific set of communication requirements to ensure the correct behaviour of the proposed algorithm.

The rest of the paper is structured as follows. Section 2 presents the formal framework used in this paper. Next, in Sect. 3 we describe the proposed distributed scheme. Section 4 presents experimental results. Finally, in Sect. 5 we present the conclusions and some lines of future work.

2 Formal Framework Used in `FARTHEST`

In this section we review the framework used for specifying and testing complex systems [MN15] that has been used to model and specify `FARTHEST`. In addition, we introduce some extensions to the finite state machine model, that allows define and check the correctness of communications between components of the system.

2.1 Finite State Machines

Finite State Machines, in short `FSM`, are one of the formalism widely used to formally specify systems. We have chosen them to specify our system because they are well known are their definition and semantics are very simple.

Definition 1. *A Finite State Machine* is a tuple $M = (\mathcal{S}, s_{in}, \mathcal{I}, \mathcal{O}, \mathcal{T}_r)$ where $\mathcal{S}$ is a finite set of states, s_{in} is the initial state of the machine, $\mathcal{I}$ is the set of input actions, $\mathcal{O}$ is the set of output actions, with $\mathcal{I} \cap \mathcal{O} = \emptyset$, and $\mathcal{T}_r$ is the set of transitions, with $\mathcal{T}_r \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{I} \times \mathcal{O}$. A transition belonging to $\mathcal{T}_r$ is a tuple (s, s', i, o) where $s, s' \in \mathcal{S}$ are the initial and final states of the transition respectively, $i \in \mathcal{I}$ is the input action and $o \in \mathcal{O}$ is the output action.

We say that M is *deterministic* if for all state s and input i there exists at most one state s' and one output o' such that $(s, s', i, o) \in \mathcal{T}_r$. We say that M is *input-enabled* if for all state s and input i there exists at least one state s' and one output o' such that $(s, s', i, o) \in \mathcal{T}_r$. □

In the following definition we introduce the concept of *trace*. A trace is a sequence of inputs and outputs pairs that captures the behaviour of a system.

Definition 2. Let $M = (\mathcal{S}, s_{in}, \mathcal{I}, \mathcal{O}, \mathcal{T}_r)$ be a `FSM`. We say that $< i_1/o_1, \ldots, i_n/o_n >$ is a *trace* of M if there exist n states $s_1, \ldots, s_n \in \mathcal{S}$ such that

$$(s_0, s_1, i_1, o_1), (s_1, s_2, i_2, o_2), \ldots, (s_{n-1}, s_n, i_n, o_n) \in \mathcal{T}_r$$

with $s_0 = s_{in}$. We denote by $<>$ the empty trace and by $\texttt{trace}(M)$ the set of all traces of M. □

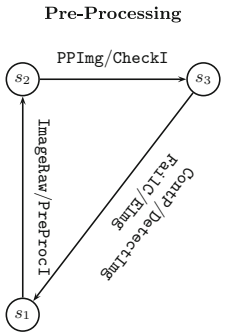


Fig. 2. Specification of an image pre-processing by using an FSM

Example 1. Let us consider the `FSM` depicted in Fig. 2 presenting a reduced version of the pre-processing image stage. It enhances the visual appearance and improves the manipulation of the image for later stages. The nodes represent the most relevant states of the algorithm, while the arcs represent the relevant transitions performed during the process. The initial state of the machine is s_1, corresponding to the point where the image to be pre-processed is received.

Let us consider the transition $(s_1, s_2, \texttt{ImageRaw}, \texttt{PreProcI})$. Intuitively, if the machine is the initial state s_1 and it receives an input `ImageRaw`, then it produces the output `PreProcI` and the machine changes to state s_2. Also, we can observe that $(\texttt{PPImg/CheckI}, \texttt{ContP/DetectImg})$ is a trace of the system.

Next, we describe the set of steps required to perform the image pre-processing. At the initial state, the system receives an image `ImageRaw` and the process responsible to handle it `PreProcI` is invoked. Once all the pre-processing operations have been performed, the system checks the correctness of the generated image calling the `CheckI` process. Finally, the checking process returns a result which shows the diagnostic of the pre-processed image. If the checking process detects that the image has some faults, then the process will be interrupted.

2.2 Communicating Finite State Machines

In order to alleviate hard computational challenges, there is a whole new generation of systems. These systems are usually distributed along the nodes of a network. Thus, the communication between the components of this network becomes a critical factor for the overall system performance. Unfortunately, the behaviour of these systems cannot be represented by using classical finite state machines and, therefore, it is required to develop new methodologies that allow us both to represent properties related to communications and to establish its correctness.

Definition 3. A *Communicating Finite State Machine*, in short `CFSM`, is a `FSM` with a set of communication channels. A *Net Communicating Finite State Machines*, in short `NETCOM`, is a pair $\mathcal{N} = (\mathcal{M}, \mathcal{C})$, where $\mathcal{M} = \{M_1, \ldots, M_n\}$ is a set of `CFSMs` such that for all $1 \leq i \leq n$ we have that $M_i = (\mathcal{S}_i, s^i_{in}, \mathcal{I}_i, \mathcal{O}_i, \mathcal{T}^i_r,)$ and $\mathcal{C} = \{\mathcal{C}^a_i : i \leq n \wedge a \in \mathcal{I}_i\}$ represents the set of communication channels, where $\mathcal{C}^a_i$ means that M_i can receive the message a. We assume that $\mathcal{I}_1, \ldots \mathcal{I}_n$ are pairwise disjoint and for all $1 \leq i \leq n$ we have that $\mathcal{I}_i \cap \mathcal{O}_i = \emptyset$.

Given $\mathcal{I}_\mathcal{N} = \bigcup_{i=1}^{n} \mathcal{I}_i$ and $\mathcal{O}_\mathcal{N} = \bigcup_{i=1}^{n} \mathcal{O}_i$, we define the sets $Shared_\mathcal{N} = \mathcal{I}_\mathcal{N} \bigcap \mathcal{O}_\mathcal{N}$, $\texttt{envInput}_\mathcal{N} = \mathcal{I}_\mathcal{N} \setminus Shared_\mathcal{N}$ and $\texttt{envOutput}_\mathcal{N} = \mathcal{O}_\mathcal{N} \setminus Shared_\mathcal{N}$. □

A `CFSM` in a `NETCOM` can interact both with the environment and with another `CFSM`, by sending inputs and receiving output actions. Thus, two classes of transitions can be distinguished. On the one hand, *external transitions* are those labelled with input actions that are received from the environment. On the other hand, *internal transitions* are those that are triggered by an output produced by the execution of a transition in another `CFSM`.

The set $\texttt{shared}_\mathcal{N}$ contains those actions allowing the communication between two machines in a net. Those actions belong simultaneously to the set of *input* actions of a `CFSM` in the net and the set of *output* actions in another one. The set $\texttt{envInput}_\mathcal{N}$ ($\texttt{envOutput}_\mathcal{N}$) corresponds to the set of *not shared* input (output) actions appearing in $\mathcal{N}$, that is, the input (output) actions labelling external transitions.

Example 2. Let us consider the `NETCOM` depicted in Fig. 3. It can be seen as an evolution of Example 1 by including communication channels. In this case, the image pre-processing, previously performed by a single `FSM` has been separated into two different `CFSM`, *Pre-Processing* and *Checking*. Moreover, we have included another `CFSM`, called *Data Base*, that provides images to the image pre-processing system.

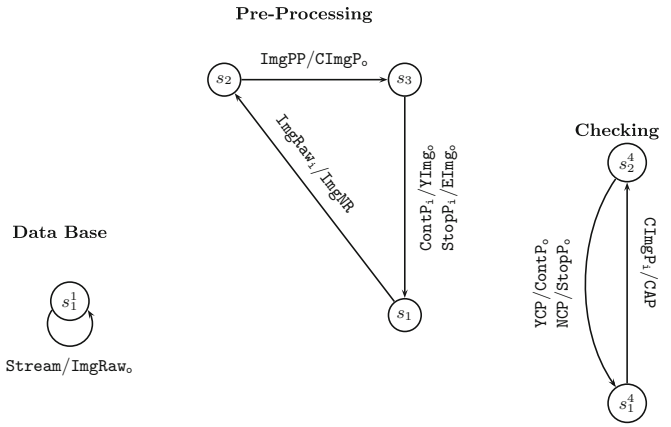


Fig. 3. Specification of pre-processing and checking image phases by using CFSM model

Next, we describe the set of steps required to perform the pre-processing and checking phases. At the initial state, the database machine provides the system with an image stream by sending the message $\texttt{ImageRaw}_\texttt{o}$ to the Pre-Processing machine. In this way, the Pre-Processing machine receives the message $\texttt{ImageRaw}_\texttt{i}$ and invokes the pre-processing operation ImgNR. Once all the pre-processing operations have been performed, the Pre-Processing machine sends the message $\texttt{CImgP}_\texttt{0}$ to Checking machine, that checks the correctness of the generated image. Finally, the checking process returns a result of the diagnostic of the pre-processed image. If the checking process detects that the image has some faults, the Checking machine sends a $\texttt{StopP}_\texttt{0}$. On the contrary, if the image is correct, it sends $\texttt{ContP}_\texttt{0}$. □

In order to validate the correctness of a system by using a passive testing technique, we record and analyse the sequences of actions generated by the system under test. These sequences are checked against a certain set of properties, that we call *invariants*, representing the most relevant properties that the system must fulfill. Next, we introduce the notion of *communication invariants*, an extension of the usual notion of invariant used in a single FSM.

Definition 4. Let $\mathcal{N} = (\mathcal{M}, \mathcal{C})$ be a NETCOM. We say that a sequence ϑ is a *communicating invariant*, in short *c-invariant*, for the net $\mathcal{N}$, if ϑ is defined according to the following EBNF:

$$\begin{aligned}
&\vartheta ::= \vartheta_1 | \vartheta_2 \\
&\vartheta_1 ::= i/s, \vartheta_2 | i/s, \vartheta_3 | i \to S \\
&\vartheta_2 ::= s/o, \vartheta_1 | s/o, \vartheta_3 | s/s', \vartheta_2 | s/s', \vartheta_3 | s \to O \\
&\vartheta_3 ::= \star, \vartheta
\end{aligned}$$

In this expression we consider $s, s' \in \texttt{shared}_\mathcal{N}$, $i \in \texttt{envInput}_\mathcal{N}$, $o \in \texttt{envOutput}_\mathcal{N}$, $S \subseteq \texttt{shared}_\mathcal{N}$ and $O \subseteq \texttt{envOutput}_\mathcal{N}$. The set of invariants for the net $\mathcal{N}$ is denoted by $\Omega_\mathcal{N}$, where we will omit the subindex if it can be deduced from the context. □

The previous EBNF expresses that a c-invariant is a sequence of symbols where each component, but the last one, is either a pair with one of the elements being a shared action (s) and the other one an input (i) or an output (o) action, or the wildcard $\star$ that can replace a sequence of actions not containing the first input symbol that appears in the component of the c-invariant that follows it. Let us note that two consecutive pairs in the sequence $a/b, c/d$ must be *compatible*, that is, either $c = b \in \texttt{shared}_{\mathcal{N}}$ or $b \in \texttt{envOutput}_{\mathcal{N}}$, $c \in \texttt{envInput}_{\mathcal{N}}$ and both a and c belong to the set of input actions of the same `CFSM` in $\mathcal{N}$. In addition, a c-invariant cannot contain two consecutive occurrences of $\star$. The last component is given by either the expression $i \rightarrow S$ or $s \rightarrow O$. The former corresponds to a input action followed by a set of shared actions and the latter represents a shared action followed by a set of output actions.

3 Distributed Schema to Detect Suspicious Artefacts

In this section we describe our proposed distributed schema for detecting suspicious artefacts, called `FARTHEST`. In order to show a detailed perspective of our approach, a formal specification of the different phases of this schema is provided. Moreover, we include a set of c-invariants to analyse the correctness of its behaviour.

`FARTHEST` can be divided into three different phases. The first phase corresponds to image pre-processing operations, the second phase ensures image integrity and the third phase performs image recognition based on majority voting process. Figures 4 and 5 show the formal specification of the `NETCOM` that represents the behaviour of `FARTHEST`, which have been developed by using the framework described in Sect. 2. First, an image to be processed ($\texttt{ImgRaw}_\texttt{i}$) is sent to the *pre-processing* phase. In this stage, the image is filtered and processed (`ImgNR`) with noise reduction algorithms to fix possible visual defects of the image. Next, the pre-processed image (`ImgPP`) is sent to the *checking* phase ($\texttt{CImgP}_\texttt{o}$), where the image ($\texttt{CImgP}_\texttt{i}$) is received and checked in order to detect format defects (`CI`). If some fault is detected, the checking phase emits a report error ($\texttt{StopP}_\texttt{o}$) and the execution is aborted ($\texttt{EImg}_\texttt{o}$). On the contrary, the image ($\texttt{VotImg}_\texttt{o}$) is sent to the *detection voting* phase. In this stage, a voting process is performed for determining the suspect nature of an element. Thus, the image received ($\texttt{VotImg}_\texttt{i}$) is sent to the detection algorithms ($\texttt{V1}_\texttt{o}, \texttt{V2}_\texttt{o}, \texttt{V3}_\texttt{o}$), where the image is processed by all of them. If the algorithm detects that the image matches with a suspicious artefact, it sends a positive vote ($\texttt{YA}_\texttt{i}$). On the contrary, it sends a negative vote ($\texttt{NA}_\texttt{i}$). Finally, if the absolute majority of the votes is positive, an alarm is triggered ($\texttt{WD}_\texttt{i}$) and the image is stored into a database ($\texttt{SaveDB}_\texttt{i}$).

In `FARTHEST`, an image stream flows through the different stages by following a pipeline model, where the output generated by a phase is considerer as the input of the next one. Inasmuch as each phase only can process one image at the same time, the execution of the phases can overlap, which allows processing multiple image detection simultaneously. Moreover, since the distributed design of `FARTHEST` allows to execute each phase in different physical machines,

the resources provided by a distributed system, like an HPC cluster or a cloud computing system, can be exploited in parallel to increase the overall system performance.

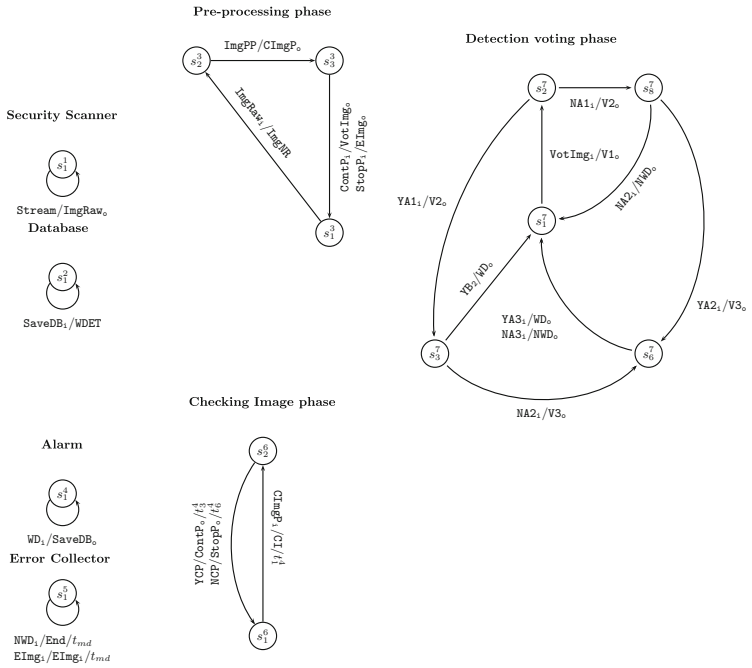


Fig. 4. Automatic weapon recognition system

We consider that the detection of elements that are a threat is critical. Thus, we provide a robust and extensible meta-detector suitable for different hazard environments, such that the detection of threat elements is performed through a voting process among several independent detection algorithms. In addition, we have included into `FARTHEST` *Online Stage* [Mer15b]. This is a detection algorithm based on the classification and analysis of the main keypoints extracted from an image.

Our algorithm can be divided into two main parts: *Monocular analysis* and *Multiple View Analysis*. Figure 5 shows the specification of this algorithm, that is represented by a `NETCOM` conformed by three `CFSM` . The first machine, *Online State*, describes the general behaviour of the algorithm, receiving an image ($\texttt{V1}_\texttt{i}$) and distributing the process among the other machines ($\texttt{MA}_\texttt{o}, \texttt{MV}_\texttt{o}$). The second machine describes the *Monocular analysis* process, performing operations such as segmentation (`SEG`), keypoints selection (`KEY`), classification and clustering (`CCS`). If the *Monocular Analysis* is correct ($\texttt{YMA}_\texttt{i}$), then the second phase of the algorithm is carried out by the machine *Multiple View Analysis*, performing operations such as data association (`DAS`) and data analysis (`DAN`). Finally, if

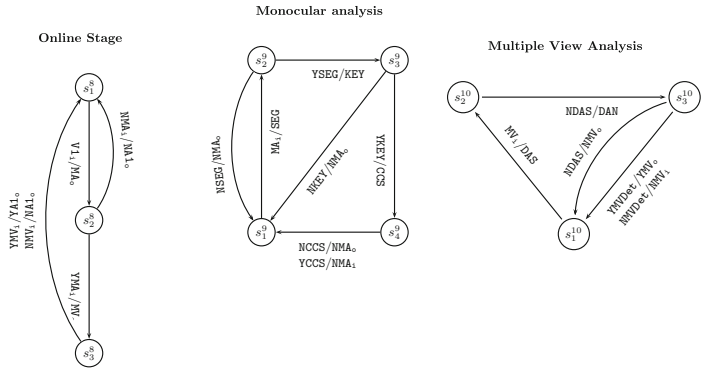


Fig. 5. Online stage algorithm

the algorithm detects a suspicious artefact, then it sends a positive vote ($\texttt{YA1}_\texttt{o}$). On the contrary, a negative vote is emitted ($\texttt{NA1}_\texttt{o}$).

Next, we include a set of communicating requirements, using c-invariants, that must be fulfilled by the implementation in order to ensure correctness.

$$\theta_1 = \underbrace{\texttt{ImgPP}}_{i} \rightarrow \underbrace{\{\texttt{VotImgB}_\texttt{o}, \texttt{CImgP}_\texttt{o}, \texttt{EImg}_\texttt{o}\}}_{S}$$

$$\theta_2 = \underbrace{\texttt{ImgRaw}_\texttt{i}/\texttt{ImgNR}}_{s/o}, \star, \underbrace{\texttt{EImg}_\texttt{i}}_{s} \rightarrow \underbrace{\{\texttt{SaveE}\}}_{O}$$

$$\theta_3 = \underbrace{\texttt{ImgRaw}_\texttt{i}/\texttt{ImgNR}}_{s/o}, \star, \underbrace{\texttt{WD}_\texttt{i}}_{s} \rightarrow \underbrace{\{\texttt{WDET}\}}_{O}$$

$$\theta_4 = \underbrace{\texttt{ImgRaw}_\texttt{i}/\texttt{ImgNR}}_{s/o}, \star, \underbrace{\texttt{NWD}_\texttt{i}}_{s} \rightarrow \underbrace{\{\texttt{End}\}}_{O}$$

4 Experiments

In this section we present several experiments to evaluate the scalability of FARTHEST when it is deployed in different cloud systems. These experiments have been conducted in a simulated environment by using the iCanCloud simulation platform [NVC+12].

Each modelled cloud contains 6 physical machines such that each phase of FARTHEST is executed in a dedicated machine and the other three machines are used for the voting process. In this model, there is one centralised database that contains the images to be processed. Each cloud where FARTHEST is deployed accesses this database through the Internet. In these experiments, FARTHEST has been deployed in 1, 2, 4 and 8 homogeneous cloud systems. Also, each experiment

uses a different configuration for the physical machines, containing 1, 2, 4 and 8 CPU cores.

The results of our experiments show that increasing the number of clouds where `FARTHEST` is deployed has a direct impact in the overall system performance, leading to a performance speed-up. However, increasing the number of CPU cores per machine slightly increases the system performance. This is mainly caused by the bottleneck located in the data base system, which hampers the exploitation of computing parallelism by using all the CPU cores at the same time.

5 Conclusions and Future Work

In this paper we have presented `FARTHEST`, a formally specified and analysed distributed schema, to detect suspicious artefacts. `FARTHEST` has been specified by using a formal framework based on Finite State Machines. Also, a set of communicating requirements to check the correct behaviour of the proposed schema has been provided. In order to show the applicability of `FARTHEST` it has been deployed along several cloud systems in a simulated environment. The experiments of this paper have been conducted by using the iCanCloud simulation platform. The evaluation results show that `FARTHEST` provides an increasing in the overall system performance when it is deployed in different cloud systems. However, since all the images are stored in a centralised data base, the communication network to access the data base acts as a bottleneck, which leads to a performance loss.

A first line of future work consists in the inclusion of timed and probabilistic information in our models [HM09,HMN09,AMN12]. We would also like to use passive testing techniques to check the proposed schema by using more complex communicating requirements. A third line of work consists in studying optimizations to reduce energy consumption [CNLC13,CNNP15] and to increase performance due to parallelization [NFM13,NM14]. Finally, we would like to use learning techniques to improve the performance of our detection algorithms taking into account that an *attacker* might modify some of the components [LNRR02].

References

[10701] US PUBLIC LAW 107–71. Aviation and transportation security act. (2001)

[AMN12] Andrés, C., Merayo, M.G., Núñez, M.: Formal passive testing of timed systems: Theory and tools. Softw. Test. Verification Reliab. **22**(6), 365–405 (2012)

[CHN15] Cavalli, A.R., Higashino, T., Núñez, M.: A survey on formal active and passive testing with applications to the cloud. Ann. Telecommun. **70**(3–4), 85–93 (2015)

[CNLC13] Castañé, G., Núñez, A., Llopis, P., Carretero, J.: E-mc^2: A formal framework for energy modelling in cloud computing. Simul. Model. Pract. Theor. **39**, 56–75 (2013)

[CNNP15] Cañizares, P.C., Núñez, A., Núñez, M., Pardo, J.J.: A methodology for designing energy-aware systems for computational science. In: 15th International Conference on Computational Science, ICCS 2015, Procedia Computer Science 51, pp. 2804–2808. Elsevier (2015)

[HM09] Hierons, R.M., Merayo, M.G.: Mutation testing from probabilistic and stochastic finite state machines. J. Syst. Softw. **82**(11), 1804–1818 (2009)

[HMN09] Hierons, R.M., Merayo, M.G., Núñez, M.: Testing from a stochastic timed system with a fault model. J. Logic Algebraic Program. **78**(2), 98–115 (2009)

[LNRR02] López, N., Núñez, M., Rodríguez, I., Rubio, F.: Including malicious agents into a collaborative learning environment. In: Cerri, S.A., Gouardéres, G., Paraguaçu, F. (eds.) ITS 2002. LNCS, vol. 2363, pp. 51–60. Springer, Heidelberg (2002)

[LW08] Liu, D., Wang, Z.: A united classification system of x-ray image based on fuzzy rule and neural networks. In: 3rd International Conference on Intelligent System and Knowledge Engineering, ISKE 2008, vol. 1, pp. 717–722. IEEE Computer Society (2008)

[Mer15a] Mery, D.: Applications in X-ray testing. In: Computer Vision for X-Ray Testing, pp. 267–325. Springer, Heidelberg (2015)

[Mer15b] Mery, D.: Inspection of complex objects using multiple-X-ray views. IEEE/ASME Trans. Mechatron. **20**(1), 338–347 (2015)

[MN15] Merayo, M.G., Núñez, A.: Passive testing of communicating systems with timeouts. Inf. Softw. Technol. **64**, 19–35 (2015)

[MR95] Murray, N.C., Riordan, K.: Evaluation of automatic explosive detection systems. In: 29th Annual International Carnahan Conference on Security Technology, pp. 175–179. Institute of Electrical and Electronics Engineers (1995)

[NFM13] Núñez, A., Filgueira, R., Merayo, M.G.: SANComSim: A scalable, adaptive and non-intrusive framework to optimize performance in computational science applications. In: 13th International Conference on Computational Science, ICCS 2013, Procedia Computer Science 18, pp. 230–239. Elsevier (2013)

[NM14] Núñez, A., Merayo, M.G.: A formal framework to analyze cost and performance in Map-Reduce based applications. J. Comput. Sci. **5**(2), 106–118 (2014)

[NPA08] Nercessian, S., Panetta, K., Agaian, S.: Automatic detection of potential threat objects in x-ray luggage scan images. In: IEEE Conference on Technologies for Homeland Security, pp. 504–509. IEEE Computer Society (2008)

[NVC+12] Núñez, A., Vázquez-Poletti, J.L., Caminero, A.C., Castañé, G.G., Carretero, J., Llorente, I.M.: iCanCloud: A flexible and scalable cloud infrastructure simulator. J. Grid Comput. **10**(1), 185–209 (2012)

[US15] Uroukov, I., Speller, R.: A preliminary approach to intelligent X-ray imaging for baggage inspection at airports. Signal Process. Res. **4**, 1–11 (2015)

[WB12] Wells, K., Bradley, D.A.: A review of X-ray explosives detection techniques for checked baggage. Appl. Radiat. Isot. **70**(8), 1729–1746 (2012)

7.8 FORTIFIER: A FORmal disTrIbuted Framework to Improve the dEtection of thReatening objects in baggage

7.8	
Publication data	
Authors:	Pablo C. Cañizares, Mercedes G. Merayo and Alberto Núñez
Title:	FORTIFIER: a FORmal disTrIbuted Framework to Improve the dEtection of thReatening objects in baggage
Publication:	Journal of Information Telecommunication
Pub. Type:	Journal
Year:	2018
DOI/URL:	https://doi.org/10.1080/24751839.2017.1347766
Pages:	17
Category:	Computer Science
Quartile:	N.A.
Ranking:	N.A.
Impact factor:	N.A.
Contribution	
Summary:	In this paper we present FORTIFIER, a formal distributed framework designed to detect suspicious artefacts. The main core of our proposed framework for recognising suspicious artefacts is divided in different phases, where each one is modelled with a specific FSM. Several FSMs are combined to detect different artefacts
Technique:	Formal Modelling
Secondary techniques:	Simulation, Cloud Computing

JOURNAL OF INFORMATION AND TELECOMMUNICATION, 2018
VOL. 2, NO. 1, 2–18
https://doi.org/10.1080/24751839.2017.1347766

FORTIFIER: a FORmal disTrIbuted Framework to Improve the dEtection of thReatening objects in baggage

Pablo C. Cañizares, Mercedes G. Merayo and Alberto Núñez

Departamento de Sistemas Informáticos y Computació, Universidad Complutense de Madrid, Madrid, Spain

ABSTRACT

Currently, security breaches in public places like airports and official buildings are a major concern by both governmental and corporative organizations. In these situations, X-ray devices must scan a vast amount of baggage in a short time frame. Hence, deploying scanners that automatize the task of detecting suspicious artefacts becomes of vital importance to prevent from threats. In this paper we present FORTIFIER, a formal distributed framework designed to detect suspicious artefacts. This approach consists in the integration of several image detection algorithms executed in a distributed environment, which are aimed to detect a wide spectrum of weapons like guns, knifes and bombs. The main core of our proposed framework for recognizing suspicious artefacts is divided in different phases, where each one is modelled with a specific finite state machine (FSM). Several FSMs are combined to detect different artefacts. We also present a case of study where some performance experiments are carried out for analysing the scalability of FORTIFIER. Initially, FORTIFIER is deployed in a single cloud environment. Once the main features that have a significant impact on the overall system performance are analysed, our proposed framework is deployed in a multi-cloud environment.

ARTICLE HISTORY
Received 5 May 2017
Accepted 25 May 2017

KEYWORDS
Formal modelling; image recognition; cloud systems

1. Introduction

The manual detection of suspicious artefacts carried out by human operators is a complex and arduous task. The luggage usually consists of a large quantity of items, usually overlapped, which hampers the detection of this kind of malicious objects. Also, the quantity of baggage that contain threatening objects represents a very low percentage of the total, and the technological support provided to the operators is very reduced. Therefore, at peak times, the worker must instantly decide whether or not a luggage can be considered as suspicious. Since each operator must analyse several baggages, the human error level can be very high even if they have been intensively trained.

After the 11/9 tragedy, security has become a national priority for governments around the world. Consequently, high security measures have been imposed on both governmental and corporative facilities to ensure the integrity of citizens. It is worth emphasizing the

CONTACT Pablo C. Cañizares pablocc@ucm.es

case of the airports, where only the U.S. government generated an annual $700 million market (107-71 2001) in the deployment of *Explosives Detection Systems* (EDS) (Murray & Riordan, 1995; Singh & Singh, 2003), which are based on X-ray imaging for scanning baggage (see Figure 1).

For all of these reasons, the massive deployment of EDS has arisen the interest of the scientific community. Several automatic detection techniques have been reported in the literature (Mery, 2015a; Wells & Bradley, 2012), such as artificial neural networks (Liu & Wang, 2008; Singh & Singh, 2004), support vector machines (SVM) (Franzel, Schmidt, & Roth, 2012; Nercessian, Panetta, & Agaian, 2008) and novel multiple-view approaches (Mery, 2015b; Mery, Riffo, Zuccar, & Pieringer, 2017; Uroukov & Speller, 2015). All these techniques, especially the multiple-view ones (see Figure 2), have high detection rates for processing all type of threatening artefacts such as guns, weapons, bombs and knifes, which represents a beneficial contribution to protect the human life. However, these techniques are designed to be executed in a single machine. Unfortunately, this solution provides a low performance, lacks of fault tolerance and it is overexposed to risks such as computer attacks.

During the last years, cloud computing systems are increasing their role due to the wide adoption of new computer networks and the fast evolution of computing technologies. Cloud computing can be defined as a paradigm that provides access to a flexible and on-demand computing infrastructure, by allowing the user to deploy virtual machines for a specific time slot giving the illusion of unlimited resources. A very clear proof of this fact is that very important companies like Amazon, Google, Dell, IBM, and Microsoft are investing billions of dollars in order to provide their own cloud solutions.

Currently, several factors motivate the interest of migrating and deploying systems in the cloud, like the possibility of accessing to a flexible computing system with the possibility of varying the number of CPUs and the memory size in runtime. The leap from private data-centres and local clusters to this new paradigm has been imposed in many research areas, it being the main reason for the evolution of computational needs. Also, the lack of infrastructure and administration duties results in ease off management and

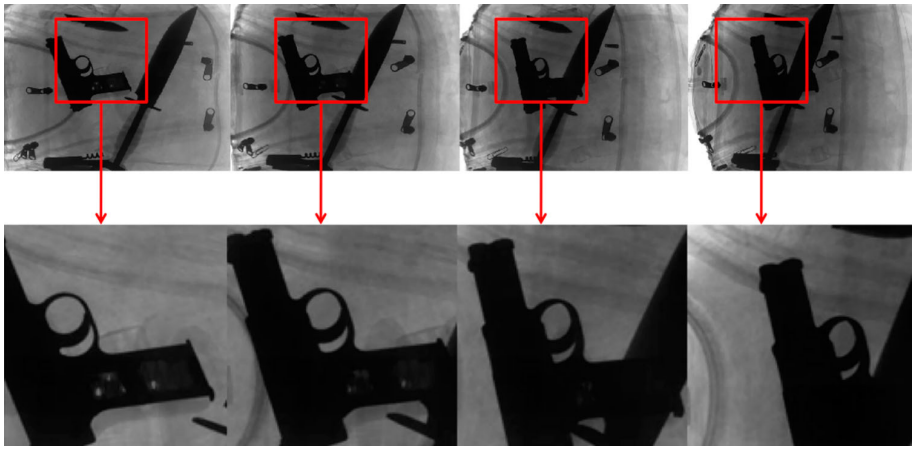


Figure 1. X-ray image for detecting guns in baggage (Mery, 2015b).

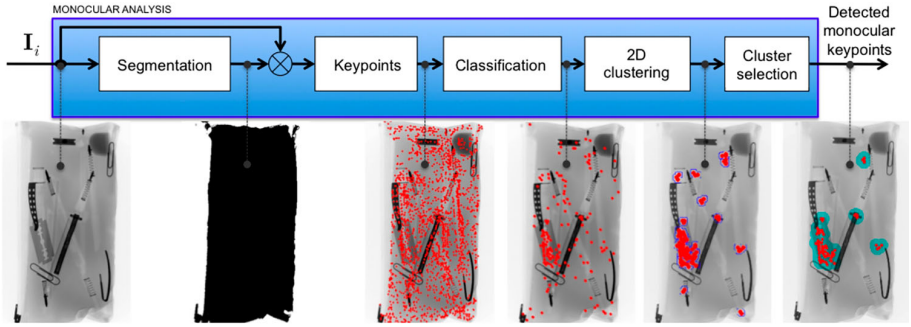


Figure 2. Excerpt of multiple-view algorithm extracted from Mery (2015b).

overall cost reductions, where users are released from those tasks for managing physical servers or storage devices.

In this paper we introduce a distributed framework, called FORTIFIER, to detect suspicious artefacts. In order to avoid security risks, it is important to incorporate mechanisms to increase the confidence on the correctness of the system. It is widely recognized that the combination of formal methods and testing techniques is very beneficial (Cavalli, Higashino, & Núñez, 2015; Gaudel, 1995; Hierons et al., 2009; Hierons, Bowen, & Harman, 2008; Hierons, Merayo, & Núñez, 2016, 2017; Veanes et al., 2008) and industry is becoming aware of the importance of using formal approaches (Grieskamp, Kicillof, Stobie, & Braberman, 2011). The system proposed in this work present interconnected components and requires a framework that allow to analyse the correctness of the communication among them. We have adopted a formal approach that allows to model our system using a formalism based on communicating finite state machines (Merayo & Núñez, 2015). Each component of FORTIFIER is given by a finite state machine. In order to ensure the correct behaviour of the proposal, a specific set of properties that involve the communication among the components has been designed and checked against our model. A complete case study has been carried out. The system has been implemented and deployed along several cloud systems in a simulated environment conducted by using the SIMCAN simulation platform (Núñez, Fernández, Filgueira, García, & Carretero, 2012). It has allowed us to analyse both the correctness of the implementation and the performance scalability of FORTIFIER.

This paper extends and enhances our previous work (Cañizares, Merayo, & Núñez, 2016). Specifically, we can mention the following contributions.

- We have included an extensive review of the main proposals to detect suspicious artefacts.
- We have designed a new set of communicating invariants that represent different behaviours that the system must fulfill.
- We have extended the evaluation process with new experiments. We report a detailed analysis of the overall system performance obtained by the application of our proposal, based on the results.

The rest of the paper is structured as follows. Section 2 reviews the related work. Section 3 presents the formal framework used in this paper. Next, in Section 4 we describe the

proposed distributed scheme. Section 5 presents experimental results. Finally, in Section 6 we present the conclusions and some lines of future work.

2. Related work

During the last years, several contributions focusing the detection of suspicious artefacts, by analysing X-ray images of luggage, can be found in the literature. In general, these artefacts could be threatening for the humans health, which is a general concern. The existing contributions can be categorized in two main groups: single-view (Liu & Wang, 2008; Singh & Singh, 2004) and multiple-view (Mery, 2015b; Uroukov & Speller, 2015).

The single-view techniques are those whose main goal is to detect threatening objects by analysing a single image. In this field, there exists a large quantity of contributions (Liu & Wang, 2007; Paranjape, Sluser, & Runtz, 1998; Turcsany, Mouton, & Breckon, 2013). Among them, let us remark two main subsets, those based on artificial neural network and those based on SVM. In the field of the artificial neural networks, Singh & Singh (2004) presented a methodology for optimizing image segmentation algorithms in an automatic way. The proposed methodology uses several image properties, such as intra-cluster distances, colour purity and gradient strength to train an artificial neural network that is used to predict the acceptance degree of the proposed solution. Finally, the authors performs an empirical study that shows the suitability of the proposal. At the same research line, Liu & Wang (2008) proposed a classification system based on artificial neural networks and fuzzy logic to detect explosives in X-ray images. Then, a multi-level fuzzy classifier and a parallel artificial neural network are used to improve the accuracy level of the proposed system. In the field of SVMs, Nercessian et al. (2008) presented an automatic system for the detection of threatening objects in X-ray luggage images. The system uses segmentation and feature vectors, which are considered as the pillars of the artificial intelligent system. An experimental study has been include to analyse and detect handguns, which shows both the effectiveness of the system for detect this kind of suspicious objects and the suitability of the algorithm for real-time applications. Al-Qubaa & Tian (2012) presented a weapon detection system based on time and frequency extraction techniques. The main idea of the proposal is that each weapon has a unique electromagnetic fingerprint, determined by its size, shape and physical composition. The empirical study carried out in this paper shows the potential and efficiency of the system by detecting guns and non-gun objects in controlled and non-controlled environments.

In the last 5 years there have been several approaches based on a novel technique known as multiple-view (Baştan, 2015; Mery, 2013, 2014). The multiple-view techniques are those whose main goal is to detect threatening objects analysing a transition of multiple views. Franzel et al. (2012) presented an automatic object detection approach for multi-view X-ray image data. This proposal is two folded, on the one hand the system analyses the variations of the X-ray images obtained from external analysers and adapts existing appearance-based object detection approaches to the X-ray image data. In this way, the authors intends to decrease distortions and to increase the feature set. On the other hand, the system uses a multi camera detection approach to analyse single-view images and multiple-view images, which improves the effectiveness of the proposed systems using the mutual reinforcement of geometrically consistent hypotheses. The experimental phase evaluates the proposed method detecting handguns in carry-on

luggage. Mery presented a multiple-view methodology to identify and extract features of a complex artefact (2015b). The methodology is based on five steps, image acquisition, geometric model estimation, single-view detection, multiple-view detection and analysis. The experimental study that has been carried out to validate the methodology shows that this proposal outperforms the existing representative approaches existing in the state of the art. Uroukov & Speller (2015) proposed a system to detect suspicions objects during the scanning process using textural signatures to recognize a wide spectrum of materials. In this work, the authors carry out an experimental study where several images of industrial standards have been filtered using a directional Gabor-type approach and analysed using a diverse spectrum of range and orientation. In the experimental phase it was found that different materials could be characterized in terms of the frequency range and orientation of the filters. More recently, Mery et al. (2017) presented an automated multiple-view method to recognize objects with highly defined shapes and sizes. The proposed method is two folded: the first step is to analyse each view of the sequence, and the next step consist in perform the analysis using the multiple-view image. With the main objective of illustrating the suitability of the proposed method, an experimental study have been carried out in order to recognize regular objects such as clips, springs and razor blades, the results have shown a high level of accuracy.

Although the existing techniques reach a suitable level of accuracy by detecting suspicious artefacts in X-ray images, for the best of our knowledge there not exist a distributed schema based on formal methods and simulation techniques to define and analyse the intended system. For this reason, we consider that the contributions proposed in this paper are suitable for the initial stages of the distributed detection systems development, decreasing the quantity of errors due to formal nature of the framework and the post-analysis performed with the simulation tool.

3. Formal framework

In this section we review the framework used for specifying and testing complex systems (Merayo & Núñez, 2015) that has been used to model and specify `FORTIFIER`. In addition, we introduce some extensions to the finite state machine model, that allows us to define and check the correctness of communications between components of the system.

3.1. Finite state machines

Finite state machines, in short `FSM`, are one of the formalism widely used to formally specify systems. We have chosen them to specify our system because they are well known and their definition and semantics are very simple.

Definition 3.1 *An FSM* is a tuple $M = (\mathcal{S}, s_{in}, \mathcal{I}, \mathcal{O}, \mathcal{T}_r)$ where $\mathcal{S}$ is a finite set of states, s_{in} is the initial state of the machine, $\mathcal{I}$ is the set of input actions, $\mathcal{O}$ is the set of output actions, with $\mathcal{I} \cap \mathcal{O} = \emptyset$, and $\mathcal{T}_r$ is the set of transitions, with $\mathcal{T}_r \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{I} \times \mathcal{O}$. A transition belonging to $\mathcal{T}_r$ is a tuple (s, s', i, o) where $s, s' \in \mathcal{S}$ are the initial and final states of the transition, respectively, $i \in \mathcal{I}$ is the input action and $o \in \mathcal{O}$ is the output action.

We say that M is *deterministic* if for all state s and input i there exists at most one state s' and one output o' such that $(s, s', i, o) \in \mathcal{T}_r$. We say that M is *input-enabled* if for all state s and input i there exists at least one state s' and one output o' such that $(s, s', i, o) \in \mathcal{T}_r$.

In the following definition we introduce the concept of *trace*. A trace is a sequence of inputs and outputs pairs that captures the behaviour of a system.

Definition 3.2 Let $M = (\mathcal{S}, s_{in}, \mathcal{I}, \mathcal{O}, \mathcal{T}_r)$ be a `FSM`. We say that $\langle i_1/o_1, \ldots, i_n/o_n \rangle$ is a *trace* of M if there exist n states $s_1, \ldots, s_n \in \mathcal{S}$ such that

$$(s_0, s_1, i_1, o_1), (s_1, s_2, i_2, o_2), \ldots, (s_{n-1}, s_n, i_n, o_n) \in \mathcal{T}_r$$

with $s_0 = s_{in}$. We denote by $\langle\rangle$ the empty trace and by `trace`(M) the set of all traces of M.

Example 3.3 Let us consider the `FSM` depicted in Figure 3 presenting a reduced version of the pre-processing image stage. It enhances the visual appearance and improves the manipulation of the image for later stages. The nodes represent the most relevant states of the algorithm, while the arcs represent the relevant transitions performed during the process. The initial state of the machine is s_1, corresponding to the point where the image to be pre-processed is received.

Let us consider the transition (s_1, s_2, `ImageRaw`, `PreProcI`). Intuitively, if the machine is the initial state s_1 and it receives an input `ImageRaw`, then it produces the output `PreProcI` and the machine changes to state s_2. Also, we can observe that (`ImageRaw/PreProcI`, `PPImg/CheckI`, `ContP/DetectImg`) is a trace of the system.

Next, we describe the steps required to perform the image pre-processing phase. At the initial state, the system receives an image `ImageRaw` and the process `PreProcI` is invoked. Once all the pre-processing operations have been performed, the system checks the correctness of the generated image calling the `CheckI` process. Finally, the checking process returns a result which shows the diagnostic of the pre-processed image. If the checking process detects that the image has some faults, then the process will be interrupted.

3.2. Communicating finite state machines

In order to alleviate hard computational challenges, there is a whole new generation of systems. These systems are usually distributed along the nodes of a network. Thus, the communication between the components of this network becomes a critical factor for

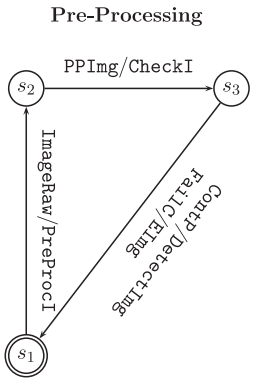


Figure 3. Specification of an image pre-processed by using an FSM.

the overall system performance. Unfortunately, the behaviour of these systems cannot be represented by using classical finite state machines and, therefore, it is required to develop new methodologies that allow us both to represent properties related to communications and to establish its correctness.

Definition 3.4 A *Communicating Finite State Machine*, in short CFSM, is an FSM with a set of communication channels. A *Net Communicating Finite State Machines*, in short NETCOM, is a pair $\mathcal{N} = (\mathcal{M}, \mathcal{C})$, where $\mathcal{M} = \{M_1, \ldots, M_n\}$ is a set of CFSMs such that for all $1 \leq i \leq n$ we have that $M_i = (\mathcal{S}_i, s^i_{in}, \mathcal{I}_i, \mathcal{O}_i, \mathcal{T}^i_r,)$ and $\mathcal{C} = \{\mathcal{C}^a_i : i \leq n \wedge a \in \mathcal{I}_i\}$ represents the set of communication channels, where $\mathcal{C}^a_i$ means that M_i can receive the message a. We assume that $\mathcal{I}_1, \ldots \mathcal{I}_n$ are pairwise disjoint and for all $1 \leq i \leq n$ we have that $\mathcal{I}_i \cap \mathcal{O}_i = \emptyset$.

Given $\mathcal{I}_\mathcal{N} = \bigcup_{i=1}^n \mathcal{I}_i$ and $\mathcal{O}_\mathcal{N} = \bigcup_{i=1}^n \mathcal{O}_i$, we define the sets $\text{Shared}_\mathcal{N} = \mathcal{I}_\mathcal{N} \bigcap \mathcal{O}_\mathcal{N}$, $\texttt{envInput}_\mathcal{N} = \mathcal{I}_\mathcal{N} \setminus \text{Shared}_\mathcal{N}$ and $\texttt{envOutput}_\mathcal{N} = \mathcal{O}_\mathcal{N} \setminus \text{Shared}_\mathcal{N}$.

A CFSM in a NETCOM can interact both with the environment and with another CFSM , by sending inputs and receiving output actions. Thus, two classes of transitions can be distinguished. On the one hand, *external transitions* are those labelled with input actions that are received from the environment. On the other hand, *internal transitions* are those that are triggered by an output produced by the execution of a transition in another CFSM.

The set $\texttt{shared}_\mathcal{N}$ contains those actions allowing the communication between two machines in a net. Those actions belong simultaneously to the set of *input* actions of a CFSM in the net and the set of *output* actions in another one. The set $\texttt{envInput}_\mathcal{N}$ ($\texttt{envOutput}_\mathcal{N}$) corresponds to the set of *not shared* input (output) actions appearing in $\mathcal{N}$, that is, the input (output) actions labelling external transitions.

Example 3.5 Let us consider the NETCOM depicted in Figure 4. It can be seen as an evolution of Example 3.3 in which we have included communication channels. In this case, the image pre-processing, previously performed by a single FSM, has been split

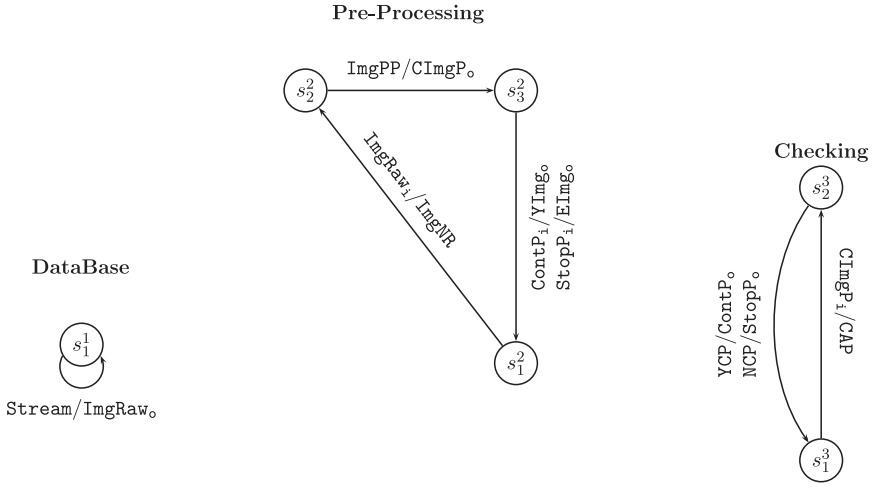


Figure 4. Specification of pre-processing and checking image phases by using the CFSM model.

between two CFSM, *pre-processing* and *checking*. Moreover, we have included another CFSM, called *database*, that provides images to the image pre-processing system.

Next, we describe the steps to perform the pre-processing and checking phases required for the analysis of images. The *database* is in charge of providing the *pre-processing* node with an image stream ($\texttt{ImageRaw}_{\texttt{o}}$). The *pre-processing* node receives it ($\texttt{ImageRaw}_{\texttt{i}}$) and invokes the process which preprocesses the image (ImgNR). Once the pre-processed image is returned (ImgPP), the *pre-processing* node sends it ($\texttt{CImgP}_{\texttt{o}}$) to the *checking* node ($\texttt{CImgP}_{\texttt{i}}$) which invokes the process that checks the correctness of the generated image (CAP). Finally, when the verdict of the evaluation is received (YCP or NCP), the *checking* node sends it ($\texttt{ContP}_{\texttt{o}}$ or $\texttt{StopP}_{\texttt{o}}$) to the *pre-processing* node. If the image presents any faults ($\texttt{StopP}_{\texttt{i}}$), the *pre-processing* node stops the process and reports an error. If the image is correct ($\texttt{ContP}_{\texttt{o}}$) the process can continue.

In order to validate the correctness of a system by using a passive testing technique, we record and analyse the sequences of actions generated by the system under test. These sequences are checked against a certain set of properties, that we call *invariants*, representing the most relevant properties that the system must fulfill. Next, we introduce the notion of *communication invariant*, an extension of the usual notion of invariant used in a single FSM.

Definition 3.6 Let $\mathcal{N} = (\mathcal{M}, \mathcal{C})$ be a NETCOM. We say that a sequence *?* is a *communicating invariant*, in short *c-invariant*, for the net $\mathcal{N}$, if *?* is defined according to the following EBNF:

$$\begin{gathered}\vartheta ::= \vartheta_1 \mid \vartheta_2 \\ \vartheta_1 ::= i/s,\, \vartheta_2 \mid i/s,\, \vartheta_3 \mid i \to S \\ \vartheta_2 ::= s/o,\, \vartheta_1 \mid s/o,\, \vartheta_3 \mid s/s',\, \vartheta_2 \mid s/s',\, \vartheta_3 \mid s \to O \\ \vartheta_3 ::= \bigstar,\, \vartheta,\end{gathered}$$

where $s, s' \in \texttt{shared}_{\mathcal{N}}$, $i \in \texttt{envInput}_{\mathcal{N}}$, $o \in \texttt{envOutput}_{\mathcal{N}}$, $S \subseteq \texttt{shared}_{\mathcal{N}}$ and $O \subseteq \texttt{envOutput}_{\mathcal{N}}$. The set of invariants for the net $\mathcal{N}$ is denoted by $\Omega_{\mathcal{N}}$, where we will omit the subindex if it can be deduced from the context.

The previous EBNF expresses that a c-invariant is a sequence of symbols where each component, but the last one, is either a pair with one of the elements being a shared action (s) and the other one an input (i) or an output (o) action, or the wildcard $\star$ that can replace a sequence of actions not containing the first input symbol that appears in the component of the c-invariant that follows it. Let us note that two consecutive pairs in the sequence $a/b, c/d$ must be *compatible*, that is, either $c = b \in \texttt{shared}_{\mathcal{N}}$ or $b \in \texttt{envOutput}_{\mathcal{N}}$, $c \in \texttt{envInput}_{\mathcal{N}}$ and both a and c belong to the set of input actions of the same CFSM in $\mathcal{N}$. In addition, a c-invariant cannot contain two consecutive occurrences of $\star$. The last component is given by either the expression $i \to S$ or $s \to O$. The former corresponds to an input action followed by a set of shared actions and the latter represents a shared action followed by a set of output actions.

A c-invariant contains two different components. The first one, called preface, includes a sequence of pairs of input/output actions in which one of them must correspond to a communication action between two machines. The second component represents the behaviour that the system must exhibit if we observe the sequence of communicating actions expressed in the preface. If we observe the preface but the next action produced

by the system is not included in the last component of the c-invariant then an error will have been detected.

4. Distributed framework to detect suspicious artefacts

In this section we describe our proposed distributed framework, called FORTIFIER, for detecting suspicious artefacts. In order to show a detailed perspective of our approach, a formal specification of the different phases of this framework is provided. We also present a set of c-invariants for analysing the efficiency and effectiveness of our proposal.

Figures 5 and 6 show the formal specification of the NETCOM that represents the behaviour of FORTIFIER. This specification has been developed using the formalism described in Section 3. It represents the different steps that constitute the whole process used for detecting suspicious artefacts that have been implemented in FORTIFIER. We can distinguish three different phases. The first one corresponds to image pre-processing operations, the second one ensures image integrity and the third one performs image recognition based on majority voting process. First, an image to be processed ($\texttt{ImgRaw}_\texttt{i}$) is sent to the *pre-processing* node. The image is filtered and processed (ImgNR) with noise reduction algorithms to fix possible visual defects of the image. Next, the pre-processed image (ImgPP) is sent to the *checking* node ($\texttt{CImgP}_\texttt{o}$), where the image ($\texttt{CImgP}_\texttt{i}$) is received and checked in order to detect format defects (CI). If

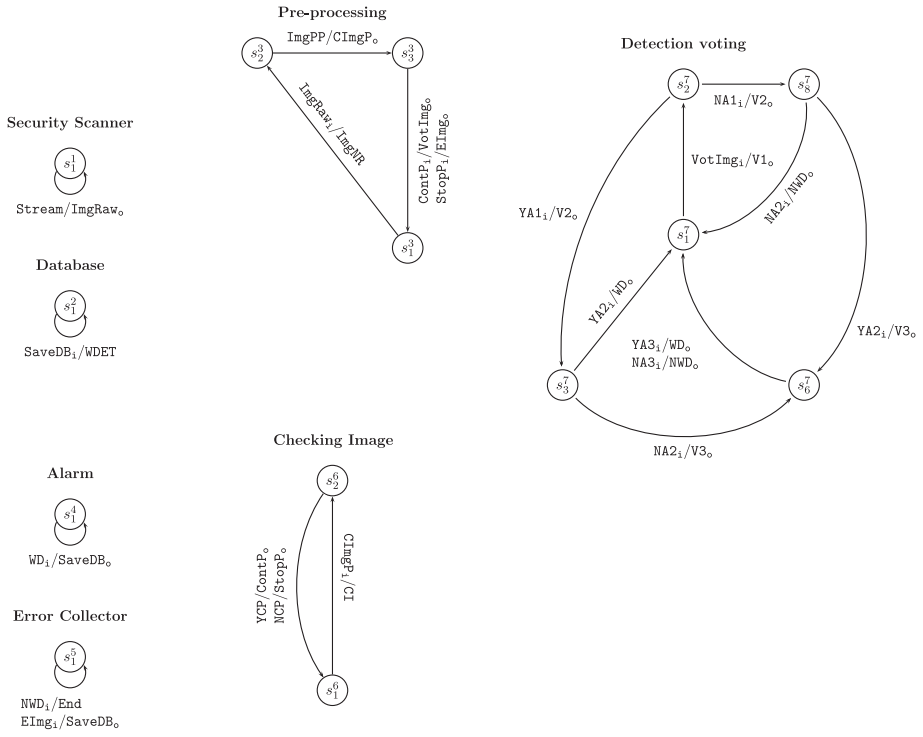


Figure 5. Automatic weapon recognition system.

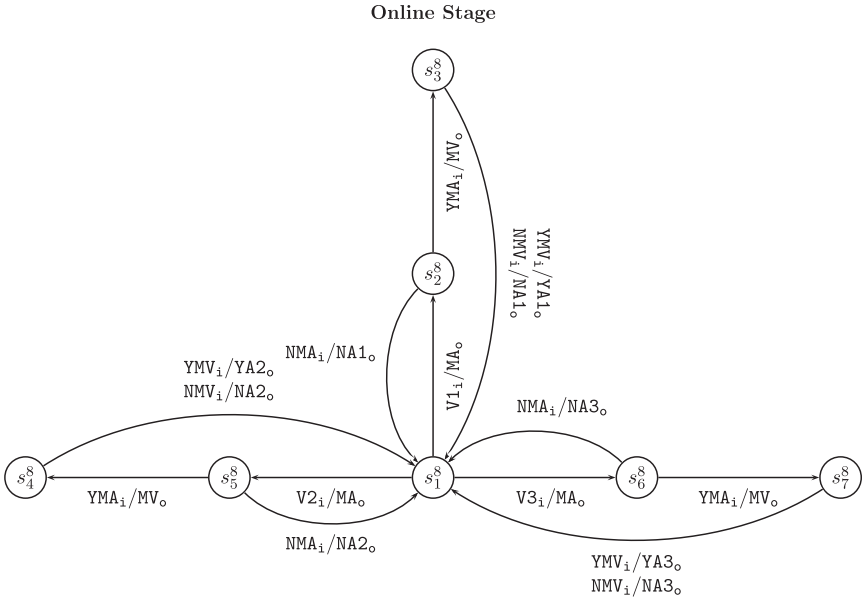


Figure 6. Online Stage Algorithm 1.

some fault is detected, the *checking* node reports an error ($\texttt{StopP}_o$) and the execution is aborted ($\texttt{EImg}_o$). If the image is correct, the system sent it to the *detection voting* node ($\texttt{VotImg}_o$). The voting process is performed for determining the suspect nature of an element. The image received ($\texttt{VotImg}_i$) is sent to the detection algorithms that process it ($\texttt{V1}_o, \texttt{V2}_o, \texttt{V3}_o$). If the algorithm detects that the image matches with a suspicious artifact, it emits a positive vote ($\texttt{YA}_i$). On the contrary, it emits a negative vote ($\texttt{NA}_i$). Finally, if the majority of the votes is positive, an alarm is triggered ($\texttt{WD}_i$) and the image is stored into a database ($\texttt{SaveDB}_i$).

In FORTIFIER, an image stream flows through the different stages by following a pipeline model, where the output generated by a node is the input of the one that performs the next step. Although each node only can process one image at the same time, the processing of different images can be performed simultaneously. Moreover, since the distributed design of FORTIFIER allows to execute the nodes in different physical machines, the resources provided by a distributed system, like an HPC cluster or a cloud computing system, can be exploited in parallel to increase the overall system performance.

We consider that the detection of elements that can be a threat to the security is critical. Thus, we provide a robust and extensible meta-detector suitable for different hazard environments, such that the detection of threatening elements is performed through a voting process in which participate several independent detection algorithms. In addition, we have included into FORTIFIER *Online Stage* (Mery, 2015b), a detection algorithm based on the classification and analysis of the main key points detected in an image.

Figures 6 and 7 show the specification of FORTIFIER, that is represented by a NETCOM with three CFSM. The first one, *Online State*, describes the general behaviour of the

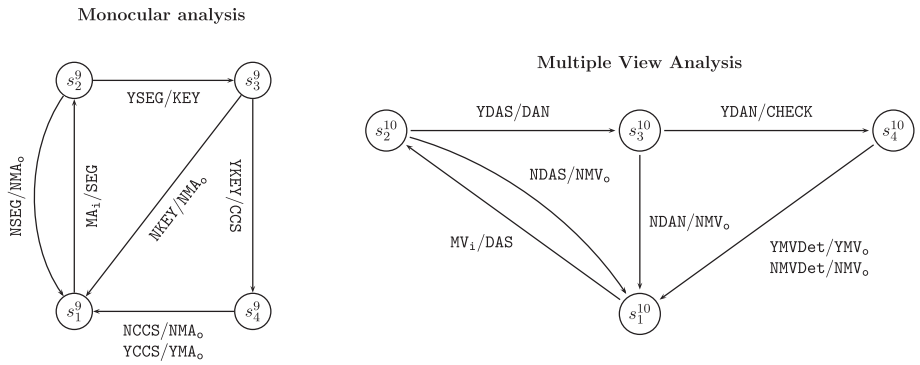


Figure 7. Online Stage Algorithm 2.

algorithm, receiving an image ($\texttt{V1}_\texttt{i}$, $\texttt{V2}_\texttt{i}$ or $\texttt{V3}_\texttt{i}$) and distributing tasks among the other machines ($\texttt{MA}_\texttt{o}$, $\texttt{MV}_\texttt{o}$). The second machine represents the behaviour of the *Monocular analysis* process, that is in charge of performing operations such as segmentation (SEG), key points selection (KEY), classification and clustering (CCS). The third machine, *Multiple View Analysis*, performs operations such as data association (DAS) and data analysis (DAN). A positive verdict ($\texttt{YA1}_\texttt{o}$) will be emitted by the algorithm if a suspicious artefact has been detected.

Next, we introduce a set of c-invariants. They represent behaviours that must be fulfilled by the system in order to ensure its correctness.

$$\theta_1 = \underbrace{\texttt{NCP}}_{i} \rightarrow \underbrace{\{\texttt{Stop}_\texttt{o}\}}_{S}$$

Intuitively, this c-invariant expresses that if the checking node detects any fault in an image (NCP), the system stops the process ($\texttt{Stop}_\texttt{o}$) of analysis of this image.

$$\theta_2 = \underbrace{\texttt{ImgRaw}_\texttt{i}/\texttt{ImgNR}}_{s/o}, \bigstar, \underbrace{\texttt{NDAS}/\texttt{NMV}_\texttt{o}}_{i/s}, \bigstar, \underbrace{\texttt{NCCS}/\texttt{NMA}_\texttt{o}}_{i/s}, \bigstar, \underbrace{\texttt{NWD}_\texttt{i}}_{s} \rightarrow \underbrace{\{\texttt{END}\}}_{O}$$

The above c-invariant expresses that after an image is loaded for being processed ($\texttt{ImgRaw}_\texttt{i}/\texttt{ImgNR}$), if we observe that no errors are detected during the multiple view analysis and the monocular analysis of two different voting algorithms, then the process should finish (END).

$$\theta_3 = \underbrace{\texttt{ContP}_\texttt{i}/\texttt{VoteImg}_\texttt{o}}_{s/s}, \bigstar, \underbrace{\texttt{MA}_\texttt{i}/\texttt{SEG}}_{s/o} \bigstar, \underbrace{\texttt{YCCS}}_{i}, \rightarrow \underbrace{\{\texttt{YMA}_\texttt{o}\}}_{S}$$

The θ_3 c-invariant represents that after an image has been sent to the detection node for voting ($\texttt{ContP}_\texttt{i}/\texttt{VoteImg}_\texttt{o}$), if at some point the monocular analysis process detects a suspicious artefact (YCCS) then it must be notified ($\texttt{YMA}_\texttt{o}$).

$$\theta_4 = \underbrace{\texttt{ImgPP}/\texttt{CImg}_\texttt{o}, \bigstar, \texttt{ContP}/\texttt{VotImg}_\texttt{o}}_{i/s,\bigstar,i/s}, \bigstar, \underbrace{\texttt{NA3}_\texttt{i}/\texttt{NW}_\texttt{o}}_{s/s} \underbrace{\texttt{NW}_\texttt{i}}_{s} \rightarrow \underbrace{\{\texttt{END}\}}_{O}$$

The last c-invariant establishes that if an image is successfully pre-processed and checked ($\texttt{ImgPP}/\texttt{CImg}_\texttt{o}$) and the image is transferred to the detection voting, then, if the vote of

the third participant is negative, the verdict of the detection voting will be negative and the process ends.

5. Experiments

In this section we present several experiments to evaluate the scalability of FORTIFIER when it is deployed in different cloud systems. In this case, each cloud system consists of a single data-centre, which have been modelled by using the SIMCAN simulation platform (Nú nez et al., 2012). Figure 8 shows the deployment infrastructure used for these experiments. In this configuration we can differentiate two main parts. First, a centralized data base that contains the images to be processed. Second, different cloud systems that access the shared data base through the Internet using a communication network.

Each cloud system contains one or several instances of FORTIFIER, which are executed for processing the corresponding set of images that are allocated in the centralized data base. It is important to remark that each instance of FORTIFIER is totally independent. For instance, two different processes that represent the same phase, like pre-processing and checking, of different FORTIFIER instances are also treated as different processes in the simulated environment. Also, each one of these processes are executed in a dedicated CPU core. Consequently, the number of FORTIFIER instances executed in the same cloud depends of the number of CPU cores used in each physical machine. Using this configuration we increase the level of parallelism in two levels. First, intra-cloud parallelism is obtained when different processes of FORTIFIER are executed in parallel using

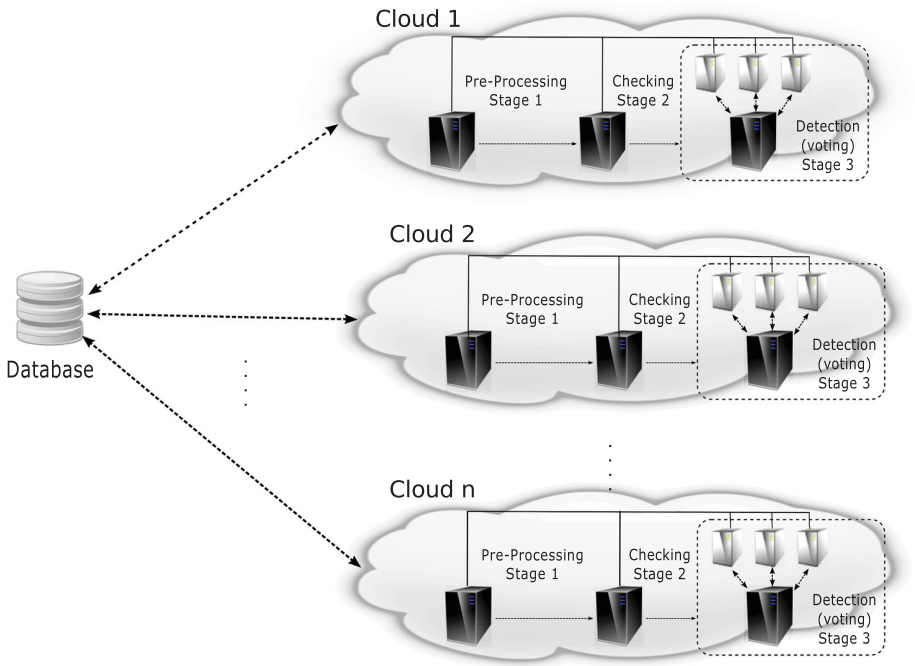


Figure 8. Deployment infrastructure of FORTIFIER.

different physical machines of the same cloud system. Second, inter-cloud parallelism is obtained when different instances of FORTIFIER are executed in parallel using several cloud systems.

It is important to note that the data representing the repository of images in the data base have been randomly generated. Thus, this data has been used to create simulated scenarios where our approach has been developed. In order to analyse the scalability of our proposed framework, FORTIFIER has been deployed in different scenarios containing 1, 2, 4 and 8 homogeneous cloud systems. It is worth to mention that all clouds have the following hardware configuration:

- CPU processor containing 1, 2, 4 and 8 CPU cores
- Hard disk drive of 1 TB.
- Network: Ethernet 100 Mbps, Ethernet Gigabit, Ethernet 10 Gbps and Ethernet 100 Gbps.
- 16 GB of RAM memory.

Initially, the system depicted in Figure 8 containing one cloud system has been modelled. In this experiment, one data-centre has been modelled using different configurations, such as the communication network and the number of CPU cores included in each physical machine. Figure 9 shows the obtained results, where the x-axis shows the type of communication network and the y-axis shows the obtained throughput, measured in processed images per minute. This chart depicts that both the communication network and the computing system act as a system bottleneck. First, the database is shared by all the physical machines and, therefore, all the images are transmitted through the same channel, which significantly decreases the overall system performance. Second, increasing the number of CPU cores per physical machines also increases the number of FORTIFIER

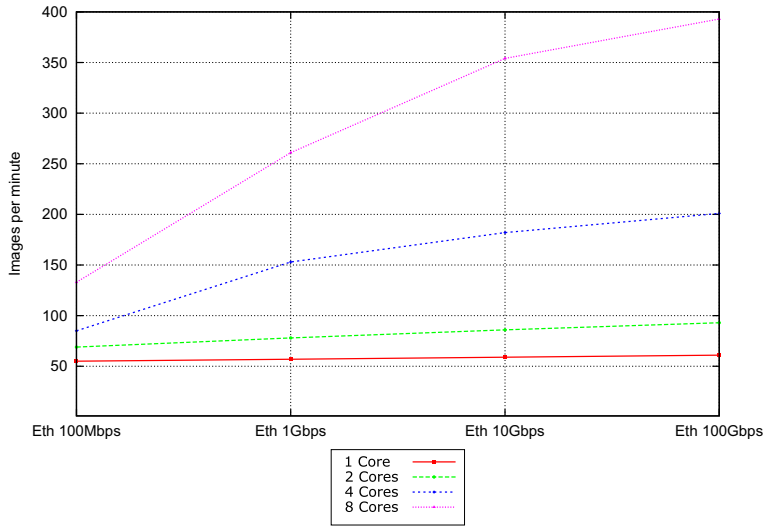


Figure 9. Performance of FORTIFIER executed in a single cloud.

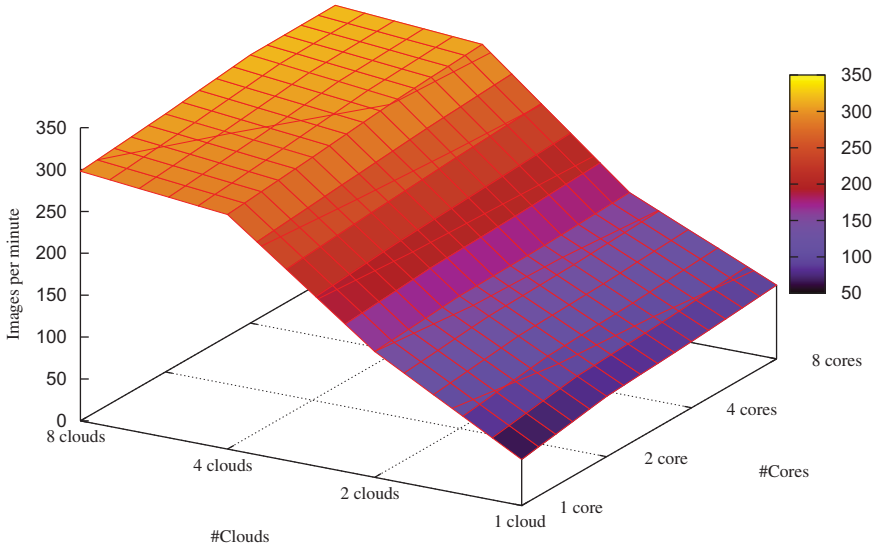


Figure 10. Performance of FORTIFIER executed in a multi-cloud environment.

instances executed in parallel. Hence, the level of parallelism for processing images is increased, which has a direct impact in the overall performance. Consequently, the best performance is obtained when both CPU resources and network bandwidth are increased.

In order to analyse the scalability of FORTIFIER when it is deployed in several cloud systems, we have modelled a scenario containing 1, 2, 4 and 8 clouds, each cloud having the same hardware configuration. This scenario uses an Ethernet 10 Gbps network. Figure 10 shows the overall performance when FORTIFIER is deployed in a multi-cloud environment, where the *x*-axis represents the number of clouds, the z-axis represents the number of CPU cores per physical machine and the *y*-axis represents the obtained throughput measured in processed images per minute.

In this case, increasing the number of clouds where FORTIFIER is deployed has a direct impact in the overall system performance, leading to a performance speed-up. However, increasing the number of CPU cores per machine slightly increases the system performance. This is mainly caused by the bottleneck located in the database system, which hampers the exploitation of computing parallelism by using all the CPU cores at the same time.

6. Conclusions and future work

In this paper we have presented FORTIFIER, a formally specified and analysed distributed framework, to detect suspicious artefacts. FORTIFIER has been specified by using a formal framework based on Finite State Machines. Also, a set of communicating requirements to check the correct behaviour of the proposed framework has been provided. In order to show the applicability of FORTIFIER it has been deployed along several cloud systems in a simulated environment. The experiments of this paper have been conducted by using the SIMCAN simulation platform. The evaluation results show that FORTIFIER provides an increasing in the overall system performance when it is deployed in

different cloud systems. However, since all the images are stored in a centralized data base, the communication network to access the data base acts as a bottleneck, which leads to a performance loss.

A first line of future work consists in the inclusion of timed and probabilistic information in our models (Andrés, Merayo, & Núñez, 2012; Hierons & Merayo, 2009; Hierons, Merayo, & Núñez, 2009). We would also like to use passive testing techniques to check the proposed framework by using more complex communicating requirements. A third line of work consists in increasing performance due to parallelization (Núñez, Filgueira, & Merayo, 2013; Núñez & Merayo, 2014). Finally, we would like to use learning techniques to improve the performance of our detection algorithms taking into account that an *attacker* might modify some of the components (López, Núñez, Rodríguez, & Rubio, 2002).

Acknowledgments

We would like to thank Manuel Núñez for useful discussions during the preparation of this paper.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

Research partially supported by the Spanish Ministerio de Economía y Competitividad projects ESTuDIo and DArDOS [TIN2012-36812-C02-01 and TIN2015-65845-C3-1-R] and the Consejería de Educación, Juventud y Deporte, Comunidad de Madrid project SICOMORo-CM [S2013/ICE-3006].

Notes on contributors

Pablo C. Cañizares is a PhD Student who received an M.Sc. degree in Computer Science in 2015 from the Universidad Complutense de Madrid. He has published 10 papers in refereed journals and international venues. His research interests are formal methods, modelling and testing of distributed systems.

Mercedes G. Merayo received her Ph.D. in Computer Science from Universidad Complutense de Madrid, Spain, in 2009. She holds an Associate Professor position in the Computer Systems and Computation Department at the same University. She has published more than 40 papers in refereed journals and international venues. Her research interests are formal methods in general, with a focus on probabilistic/timed/stochastic extensions in formal testing.

Alberto Núñez received an M.Sc. degree in Computer Science in 2005 from the Universidad Carlos III de Madrid, and a Ph.D. degree in Computer Science in 2011 from the same university. He is currently Teaching Assistant at the Universidad Complutense de Madrid, teaching Distributed Systems, Discrete Mathematics and Data Bases. He won the IBM Ph.D. Fellowship award in 2009. His research interests are focused on formal testing and high performance computing architectures and applications, especially on how to perform models and simulations.

ORCID

P. C. Cañizares http://orcid.org/0000-0002-2084-1558
M. G. Merayo http://orcid.org/0000-0002-4634-4082
A. Núñez http://orcid.org/0000-0001-8613-746X

References

107-71, U. P. L. (2001). *Aviation and transportation security act*. Washington, DC: United States Government Publishing Office.

Al-Qubaa, A., & Tian, G. (2012). Weapon detection and classification based on time-frequency analysis of electromagnetic transient images. *International Journal on Advances in Systems and Measurements*, *5*(3), 89–99.

Andrés, C., Merayo, M. G., & Núñez, M. (2012). Formal passive testing of timed systems: Theory and tools. *Software Testing, Verification and Reliability*, *22*(6), 365–405.

Baştan, M. (2015). Multi-view object detection in dual-energy X-ray images. *Machine Vision and Applications*, *26*(7–8), 1045–1060.

Cañizares, P. C., Merayo, M. G., & Núñez, A. (2016). FARTHEST: FormAl distRibuTed scHema to dEtect Suspicious arTefacts. In *8th Asian conference on intelligent information and database systems, ACIIDS'16, LNAI 9621* (pp. 770–779). Da Nang: Springer.

Cavalli, A. R., Higashino, T., & Núñez, M. (2015). A survey on formal active and passive testing with applications to the cloud. *Annales of Telecommunications*, *70*(3–4), 85–93.

Franzel, T., Schmidt, U., & Roth, S. (2012). Object detection in multi-view X-ray images. *Pattern Recognition*, *7476*, 144–154.

Gaudel, M.-C. (1995). Testing can be formal, too! In *6th int. joint conf. CAAP/FASE, theory and practice of software development, TAPSOFT'95, LNCS 915* (pp. 82–96). Aarhus: Springer.

Grieskamp, W., Kicillof, N., Stobie, K., & Braberman, V. (2011). Model-based quality assurance of protocol documentation: Tools and methodology. *Software Testing, Verification and Reliability*, *21*(1), 55–71.

Hierons, R. M., Bogdanov, K., Bowen, J., Cleaveland, R., Derrick, J., Dick, J., … Zedan, H. (2009). Using formal specifications to support testing. *ACM Computing Surveys*, *41*(2), 9.

Hierons, R. M., Bowen, J., & Harman, M. (Eds.) (2008). *Formal methods and testing, LNCS 4949*. Berlin: Springer.

Hierons, R. M., & Merayo, M. G. (2009). Mutation testing from probabilistic and stochastic finite state machines. *Journal of Systems and Software*, *82*(11), 1804–1818.

Hierons, R. M., Merayo, M. G., & Núñez, M. (2009). Testing from a stochastic timed system with a fault model. *Journal of Logic and Algebraic Programming*, *78*(2), 98–115.

Hierons, R. M., Merayo, M. G., & Núñez, M. (2016). Controllability through nondeterminism in distributed testing. In *28th IFIP WG 6.1 int. conf. on testing software and systems, ICTSS'16, LNCS 9976* (pp. 89–105). Graz, Austria.

Hierons, R. M., Merayo, M. G., & Núñez, M. (2017). An extended framework for passive asynchronous testing. *Journal of Logic and Algebraic Programming*, *86*(1), 408–424.

Liu, D., & Wang, Z. (2007). A method of X-ray image recognition based on fuzzy rule and parallel neural networks. In *International symposium on neural networks* (pp. 1231–1239). Nanjing: Springer.

Liu, D., & Wang, Z. (2008). A united classification system of X-ray image based on fuzzy rule and neural networks. In *3rd int. conf. on intelligent system and knowledge engineering, ISKE'08* (Vol. 1, pp. 717–722). Xiamen: IEEE Computer Society.

López, N., Núñez, M., Rodríguez, I., & Rubio, F. (2002). Including malicious agents into a collaborative learning environment. In *8th int. conf. on intelligent tutoring systems, LNCS 2363* (pp. 51–60). Jhongli: Springer.

Merayo, M. G., & Núñez, A. (2015). Passive testing of communicating systems with timeouts. *Information and Software Technology*, *64*, 19–35.

Mery, D. (2013). X-ray testing: The state of the art. *Journal Departement of Computer Science-Pontificia Universidad Catolica de Chile Av. Vicuna Mackenna. Santiago de Chile*, *18*(9), 1–12.

Mery, D. (2014). Computer vision technology for X-ray testing. *Insight-Non-Destructive Testing and Condition Monitoring*, *56*(3), 147–155.

Mery, D. (2015a). Applications in X-ray testing. In *Computer vision for X-ray testing* (pp. 267–325). Berlin: Springer.

Mery, D. (2015b). Inspection of complex objects using multiple-X-ray views. *IEEE/ASME Transactions on Mechatronics*, *20*(1), 338–347.

Mery, D., Riffo, V., Zuccar, I., & Pieringer, C. (2017). Object recognition in X-ray testing using an efficient search algorithm in multiple views. *Insight-Non-Destructive Testing and Condition Monitoring*, *59*(2), 85–92.

Murray, N. C., & Riordan, K. (1995). Evaluation of automatic explosive detection systems. In *29th annual int. Carnahan conf. on security technology* (pp. 175–179). Surrey: Institute of Electrical and Electronics Engineers.

Nercessian, S., Panetta, K., & Agaian, S. (2008). Automatic detection of potential threat objects in X-ray luggage scan images. In *IEEE conference on technologies for homeland security* (pp. 504–509). Waltham: IEEE Computer Society.

Núñez, A., Fernández, J., Filgueira, R., García, F., & Carretero, J. (2012). SIMCAN: A flexible, scalable and expandable simulation platform for modelling and simulating distributed architectures and applications. *Simulation Modelling Practice and Theory*, *20*(1), 12–32.

Núñez, A., Filgueira, R., & Merayo, M. G. (2013). SANComSim: A scalable, adaptive and non-intrusive framework to optimize performance in computational science applications. In *13th int. conf. on computational science, ICCS'13, procedia computer science 18* (pp. 230–239). Barcelona: Elsevier.

Núñez, A., & Merayo, M. G. (2014). A formal framework to analyze cost and performance in Map-Reduce based applications. *Journal of Computational Science*, *5*(2), 106–118.

Paranjape, R., Sluser, M., & Runtz, E. (1998). Segmentation of handguns in dual energy X-ray imagery of passenger carry-on baggage. In *Canadian conference on electrical and computer engineering* (Vol. 1, pp. 377–380). Waterloo: IEEE.

Singh, S., & Singh, M. (2003). Explosives detection systems (eds) for aviation security. *Journal of Signal Processing*, *83*(1), 31–55.

Singh, M., & Singh, S. (2004). Image segmentation optimisation for X-ray images of airline luggage. In *Proceedings of the international conference on computational intelligence for homeland security and personal safety, CIHSPS'04* (pp. 10–17). Venice: IEEE.

Turcsany, D., Mouton, A., & Breckon, T. (2013). Improving feature-based object recognition for X-ray baggage security screening using primed visual words. In *International conference on industrial technology (ICIT'13)* (pp. 1140–1145). Cape Town: IEEE.

Uroukov, I., & Speller, R. (2015). A preliminary approach to intelligent X-ray imaging for baggage inspection at airports. *Signal Processing Research*, *4*, 1–11.

Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., & Nachmanson, L. (2008). Model-based testing of object-oriented reactive systems with Spec Explorer. In *Formal methods and testing, LNCS 4949* (pp. 39–76). Berlin: Springer.

Wells, K., & Bradley, D. A. (2012). A review of X-ray explosives detection techniques for checked baggage. *Applied Radiation and Isotopes*, *70*(8), 1729–1746.

7.9 LAnt: Model Driven Approach for Ant Colony Optimization

7.9	
Publication data	
Authors:	Pablo C. Cañizares, Mercedes G. Merayo and Juan M. Vara
Title:	LAnt: Model driven approach for ant colony optimization
Publication:	Journal of Intelligent and Fuzzy Systems
Pub. Type:	Journal
Year:	2017
DOI/URL:	https://doi.org/10.3233/JIFS-169132
Pages:	11
Category:	Computer Science, Artificial Intelligence
Quartile:	Q3
Ranking:	76/132
Impact factor:	1.426
Contribution	
Summary:	This paper introduces a new high-level domain-specific modelling language, LAnt, for the design of ant colony optimization algorithms. This language is used to represent the main elements required to define the structure of the algorithms and to capture the specific constraints associated to the problem to solve. It aims to support users with low experience in the development of solutions based on this paradigm. The proposal has been implemented as an Eclipse plug-in, including an editor and an integrated code generator
Technique:	Model Driven Engineering
Secondary techniques:	Swarm Intelligence, AI

Journal of Intelligent & Fuzzy Systems 32 (2017) 1343–1354
DOI:10.3233/JIFS-169132
IOS Press

LAnt: Model driven approach for ant colony optimization[1]

Pablo C. Cañizares[a,*], Mercedes G. Merayo[a] and Juan M. Vara[b]
[a]*Universidad Complutense de Madrid, Spain*
[b]*University Rey Juan Carlos, Spain*

Abstract. This paper introduces a new high-level domain-specific modelling language, *LAnt*, for the design of ant colony optimization algorithms. This language is used to represent the main elements required to define the structure of the algorithms and to capture the specific constraints associated to the problem to solve. It aims to support users with low experience in the development of solutions based on this paradigm. The proposal has been implemented as an Eclipse plug-in, including an editor and an integrated code generator.

Keywords: Artificial intelligence, ant colony optimization, model driven engineering, domain specific language

1. Introduction

Swarm intelligence [5] helps to solve problems by means of techniques inspired in the collective behaviour of many individuals that interact among them and with the environment without any individual controlling the group. "A collective decision capability that is as least good as or better that a single decision" [22]. In some cases, the natural selection has combined thousands of individual brains to conform higher entities. This is the case of species which live in colonies, flocks, schools or crows that denotes collectively intelligent behaviour when finding paths for food or fighting off predators. This natural behaviour has been applied in computer science to design computational techniques inspired in these collective environments. Many of them, such as Bee Colony [23], Particle Swarm [24], Grey Wolf [28] or Black hole Optimization [20] are devoted to solve optimization problems. It is specially relevant the case of Ant colony optimization (ACO) that has been the basis for the development of solutions for different engineering problems, such as scheduling [29], constraint satisfaction [35] or graph coloring [9]. Besides, this technique has been used in other computational research fields. This is the case of the machine learning [11], pattern recognition [8] or data mining [30]. Furthermore, ACO paradigm has significant presence in other orthogonal disciplines such as sociology [7], cognitive science [31], biology [34] and economy [40]. The development of these solutions, in most cases, comprises thousand of code lines. The more is the length of the solution, the more is the probability of occurrence a software failure. It is well known that the 50% of a project budget is spent on finding bugs. Therefore, it is important to include mechanisms in early stages of the software development cycle that allow to detect and reduce possible errors. The use of formal methods aids to increment the software robustness and software developed using them is usually free of errors. Several formal methodologies even generate executable code. However, it requires a deep mathematical background and makes developers reluctant to apply them.

*Corresponding author. Pablo C. Cañizares, Universidad Complutense de Madrid, Spain. Tel.: +34 91 3947648; Fax: +34 91 3947529; E-mail: pablocc@ucm.es.

[1]Research partially supported by the Spanish project DArDOS (TIN2015-65845-C3-1-R) and the Comunidad de Madrid project SICOMORo-CM (S2013/ICE-3006).

7.10 Using ants to fight wildfires

7.10	
Publication data	
Authors:	Pablo C. Cañizares, Alberto Núñez and Mercedes G. Merayo
Title:	Using ants to fight wildfires
Publication:	14th International Work-Conference on Artificial Neural Networks
Pub. Type:	Conference
Year:	2017
DOI/URL:	`https://doi.org/10.1007/978-3-319-59147-6_32`
Pages:	10
CORE Ranking:	B
Contribution	
Summary:	In this work a we describe platform to deploy an algorithm based on Ant Colony Optimization to plan and attack fire focus. The framework is based on a theoretical model that allows us to represent the main elements of the environment in which fire evolves
Technique:	Evolutionary Algorithms
Secondary techniques:	Formal Modelling, Simulation, AI

Using Ants to Fight Wildfire

Pablo C. Cañizares(✉), Mercedes G. Merayo, and Alberto Núñez

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid, Madrid, Spain
{pablocc,mlmgarci,albenune}@ucm.es

Abstract. The control of fire spreading is a (research) challenge. The impact of the fire in the environment makes essential the study and analysis of fire spread with the goal of designing new tools that help to mitigate the wildfire expansion and, as a consequence, their effects. In this work we introduce a platform to deploy an algorithm, based on Ant Colony Optimization, to determine the best plan to attack fire focus. The framework is based on a theoretical model that allows us to represent the main elements of the environment in which fire evolves. The tool provides a visualisation component to model realistic landscapes.

1 Introduction

Forests play an essential role on the biological diversity. With the 9% total surface of the earth, it is known that the 50% existing species lives in forest ecosystem [11]. Therefore, forests make the planet more habitable, emitting oxygen and cleaning the air by the absorption of contamination. In addition, the wooded areas have an ecological function, preserving and regulating the climate, contributing to limit the effects of storms and acting as a heat buffer [8].

Unfortunately, an average of one percent of world's forests is reported to be affected each year by fires. The global annual area burned in the last years has reached an average of 377 Mha [9]. Wildfires can have deep impact on forest ecosystem, destroying the native species and leading to longer-term indirect effects such as loss of habitat, affecting to the nutrient retention and water infiltration. The loss of organisms, such as invertebrates, pollinators and decomposers, which are a fundamental element in forest ecosystems, can slow down the growth of the forest [13]. Another consequence of fires is the impact generated in the global CO_2 emissions and global warming. It is shown that fires play a key role in the CO_2 levels. Fires produce CO_2 emissions equal to 50% of those originated from humans [6]. Additionally, wildfires can cause several damages to houses and buildings, as well as, in extreme situations, human deaths. The forest fires in Australia (2009) and Greece (2007) caused 173 and 80 victims, respectively.

Research partially supported by the Spanish MEC project DArDOS (TIN2015-65845-C3-1-R) and the Comunidad de Madrid project SICOMORo-CM (S2013/ICE-3006).

I. Rojas et al. (Eds.): IWANN 2017, Part II, LNCS 10306, pp. 371–380, 2017.
DOI: 10.1007/978-3-319-59147-6_32

Besides to the damages caused by the fire, it is worth noting that millions of dollars are spent to extinguish and reconstruct the affected areas. A full accounting considers long-term and complex costs, including impact on watersheds, ecosystems, infrastructure, businesses, individuals, and the local and national economy. As an example, the estimated cost of a massive wildfire that devastated 125,000 acres in California (2003) came to 1.2 billion$ and required the evacuation of 100,000 residents [7].

The deep impact caused by wildfires on the ecosystem have raised the interest of the scientific community. During the last years, several approaches have been introduced to minimize the risks caused by the fire disasters. Among them, it can be found learning methodologies to build decision making systems [17], techniques based on artificial intelligence to predict the beginning of a fire focus in an specific zone using a knowledge-based system [3,15] and schedule approaches to planning the extinction [4].

In this paper we propose a forest fire modelling framework, biologically inspired, to detect and mitigate the wildfires expansion. The framework includes a flexible and scalable platform to deploy algorithms based on swarm intelligence, specifically, it includes one Ant Colony Algorithm used to optimize the detection of fire focus. We have developed a theoretical model that allows us to represent the main elements of the environment in which fire evolves. In order to simulate the expansion of the fire we have use a fire spread model based on cellular automata [10], in which all the elements of our model, affecting the forest fire spreading, can easily be incorporated. The platform has been developed using an advanced simulation engine, known as OMNeT++, that provides a visualisation platform to model realistic landscapes.

The rest of the paper is structured as follows. Section 2 presents the forest wildfire model. Next, in Sect. 3 we describe the swarm intelligence algorithm used to solve the problem. Section 4 presents some experiments. Finally, in Sect. 5, we present the conclusions and some lines of future work.

2 Forest Fire Model

One of the main goals of this paper is to provide some mechanisms which assists in detecting, preventing and mitigating the effects of wildfires in the natural ecosystems. With this aim, we have developed a framework to model, with a realistic detail level, the most important elements which are involved in a forest fire.

One of the main actors of this phenomenon are the *forests*. It is well known the wide spectrum, diversity and extension of the existing wooded areas. Therefore, it is usual that these areas are composed by several species of trees, brushwood and other geographical elements such as mountains, rivers and grasslands. In order to deal with all these aspects, we decided to model the global *surface* of a forest by a grid, in which each square corresponds to a *region*. The user can model the forest in regions which must have the same size but the dimension can range from centimeters to hectares. These regions include information related to the most

relevant characteristics of the land, such as the vegetation volume, humidity, temperature, elevation, inclination and probability of fire propagation. Another factor that must be considered, due to its relevant influence in the propagation of fires, is the evolution of *wind flows* that affect the different areas of the forest. Therefore, our model also include data corresponding to the resultant of all wind speeds and directions that act over each of the regions.

Definition 1. Let $\mathcal{G} = \{0, 1, ..., n\} \times \{0, 1, ..., m\}$ where $n, m \in \mathbb{N}$. We define a *region* of $\mathcal{G}$ as a tuple $r = (c, v, p, st, l, i, t, hum, wf)$ where $c \in \mathcal{G}$ represents the coordinates of the region, v is the total volume of vegetation measured in $\frac{m^3}{ha}$, $p \in [0, 1]$ is the probability of fire propagation in this area, st indicates the state of the region, $l \in \mathbb{R}$ is the elevation of the land measured in meters, $i \in [0, 90]$ represents the inclination angle, $t \in \mathbb{R}$ is the average temperature measured in Celsius degrees, $hum \in [0, 100]$ represents the humidity level of the area and $wf \in [0, 360) \times [0, 12]$ indicates the direction measured in degrees and the wind speed based on Beaufort scale [5] that affect the region.

The state of the region st takes values in $\{Healthy, OnFire, Burned\}$. *Healthy* means that the region is clear of fire, *OnFire* indicates that the region is partially on fire and *Burned* indicates that the fire burnt down the region to the ground. Given a region $r = (c, v, p, st, l, i, t, hum, wf)$ we let $pos(r)$ be equal to c.

A *surface* modeled by $\mathcal{G}$ is a tuple $\mathcal{S}_\mathcal{G} = (h, w, R, ld)$ where $h, w \in \mathbb{N}_+$ correspond to the height and weight of a forest measured in meters, respectively, R is a set of regions of $\mathcal{G}$ and $ld : R \times R \longrightarrow \mathbb{R}_> \cup \{\infty\}$ is the difficulty level of access function. In order to ensure that the set of regions completely covers the surface we need two conditions hold:

1. For all distinct $r, r' \in R$ we have $pos(r) \neq pos(r')$.
2. For all $c \in \mathcal{G}$ there exists $r \in R$ such that $pos(r) = c$.

Given $r_1, r_2 \in R$ such that $pos(r_1) = (x_1, y_1)$ and $pos(r_2) = (x_2, y_2)$ we say that r_1 and r_2 are *neighbors*, denoted by $neigh(r_1, r_2)$, if and only if $|x_1 - x_2| \leq 1 \wedge |y_1 - y_2| \leq 1 \wedge (x_1 = x_2 \oplus y_1 = y_2)$.

The function $ld(r_1, r_2)$ returns a value that represents the difficulty level to access between r_1 and r_2. The function returns ∞ for all the pairs (r_1, r_2) such that $neigh(r_1, r_2)$ does not hold. We say that $\sigma =< r_1, ..., r_w >$ is a *path* of $\mathcal{S}_\mathcal{G}$ if and only if for all $1 \leq i < w, neigh(r_i, r_{i+1})$ and for all $1 \leq i < j \leq w$ we have $r_i \neq r_j$. Abusing the notation we will write $r \in \sigma$ if there exists $1 \leq i \leq w$ such that $r = r_i$. □

Simulating the behavior of wildfires over a surface requires the application of wildfire models [2,12,14,16], a collection of equations that allow to calculate rate of spread, fireline intensity, fuel consumption and fire effects among others. The application of these models is indispensable for forest fire management and they are essential in the operating tools used in forestry agencies. In this work, due to the features of our approach, we have applied a model based on cellular automata (CA) [10] in which all the factors affecting the forest fire spreading can easily be incorporated. The model leads to algorithms which can exploit the

inherent parallelism of the CA structure. Cellular automata consist of a grid of cells These cells are in a specific state that changes over time on the basis of its neighbors states and a function. Despite the simplicity of this structure, it allows to model complex systems. The simulation of the wildfire corresponds to the evolution of the states of the regions define in the considered surface along the time.

3 Swarm Intelligence Algorithm

In this work we propose a parallel algorithm inspired on swarm intelligence to mitigate the effects of the wildfires in the natural ecosystems. Given a surface in which different fire focuses have been detected the algorithm searches for the shortest paths to reach each of them from a specific point where the firefighting brigade is located. In this way, we try to help to determine the best strategy for fire control. Specifically, our approach follows the Ant Colony Optimization (ACO) metaheuristic, which imitates the behaviour of real ants to solve complex problems in a distributed way. In order to apply the classical ACO paradigm to the problem faced in this work, the different features considered in our model have been integrated in the path selection and pheromone update functions.

Definition 2. Let $\mathcal{S}_\mathcal{G} = (h, w, R, ld)$ be a surface, $F \subseteq R$ be the set of the regions where the fire focuses are located and $r_0 \in R$ be the region considered the starting point. Let $\sigma_k =< r_0, r_1^k, ..., r_{l_k}^k >$ the path traversed by an ant k from the starting region. We define the probability of an ant k moving from region r_i to region r_j having as target a region $r_t \in F$ as:

$$p_{ij}^{kt} = \begin{cases} \frac{V_{ij}^t}{\sum_{neigh(r_i,r_j) \wedge r_j \notin \sigma_k} V_{ij}^t} & \text{if } r_i = r_{l_k}^k \wedge neigh(r_i, r_j) \wedge r_j \notin \sigma_k \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where $V_{ij}^t = \left[\tau_{ij}^t\right]^\alpha \cdot \left[\frac{1}{ld(r_i,r_j)}\right]^\beta$

The component τ_{ij}^t represents the pheromone value associated with the connection between the involved regions. Initially, the level of pheromone between any two regions is 0. As we previously said, the difficulty level of access function $ld(r_i, r_j)$ included in the model of the surface takes into account the different parameters associated with the regions (elevation, inclination, temperature...) that can affect the accessibility. Finally α, β are constants used to assign a weigh to each of the considered parameters.

The update of τ_{ij}^t, that indicates the pheromone level deposited by ants in the transition between regions r_i and r_j on the way to the target r_t, is performed as follows:

$$\tau_{ij}^t \leftarrow (1 - \rho) \cdot \tau_{ij}^t + \sum_{k=1}^{m} \Delta\tau_{ij}^{tk} \quad (2)$$

```
Algorithm Ants_Mitigate_Fire(focuses, nAnts, nIter, startReg)
while (termination condition does not hold) do
    Solution ← ∅;
    Targets ← focuses ∪ starReg;
    foreach origReg in Targets do
        foreach destReg in Targets do
            lIter ← nIter;
            while lIter > 0 do
                SolutionTarget ← ∅;
                Release ants;
                foreach ant do
                    Path ← origReg;
                    while (destReg not reached ∧ antIsAlive) do
                        step ← calculateNextStep();
                        if step ≠ ∅ then
                            Path ← Path ∪ step;
                        else
                            killAnt();
                        end
                    end
                    SolutionTarget ← SolutionTarget ∪ Path;
                end
                updatePheromones();
                lIter ← lIter − 1;
            end
            Solution ← Solution∪ shortestPath(origReg,destReg);
        end
    end
    buildHamiltonGraph(Solution);
    spreadFire();
end
```

Algorithm 1. Algorithm schema for mitigate wildfire inspired by ACO.

where ρ denotes the pheromone evaporation rate, m is the number of ants and $\Delta\tau_{ij}^{tk}$ is the amount of pheromone deposited by ant k in the transition from r_i to r_j, which is given by the following equation:

$$\Delta\tau_{ij}^{kt} = \begin{cases} Q/l_k & \text{if } \exists\ 1 \leq h < l_k : r_h = r_i \wedge r_{h+1} = r_j \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where Q is a parameter of the model. □

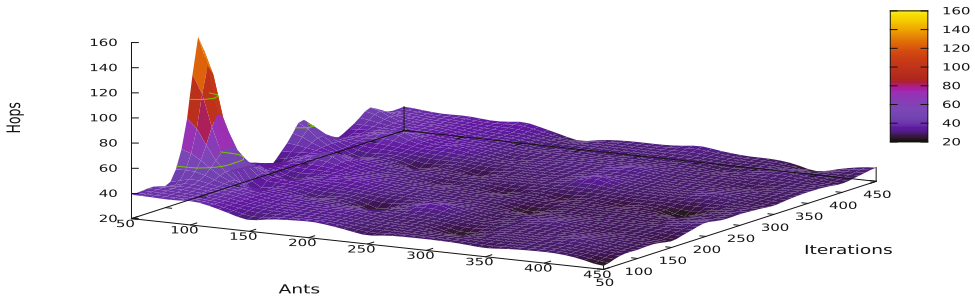
The proposed algorithm is shown in Algorithm 1. Intuitively, the algorithm calculates the shortest paths between the starting region and each fire focus. The process consists in the application of the ACO metaheuristic, using the path selection and pheromone update function previously defined. It is worth noting that if any ant is not able to continue on its way to the target focus,

this ant is killed and its local solution does not affect to the pheromone update. The path construction process is performed a predefined number of iterations until a final solution is obtained. At this point, each region containing a fire focus and the starting region are connected by a path. In order to select the shortest path that covers all the fire focuses, a hamiltonian graph is built using the provided solution. Afterwards, the fire evolves by applying the fire spreading model proposed in the previous section. This process does not stop until the predefined number of iterations have been performed.

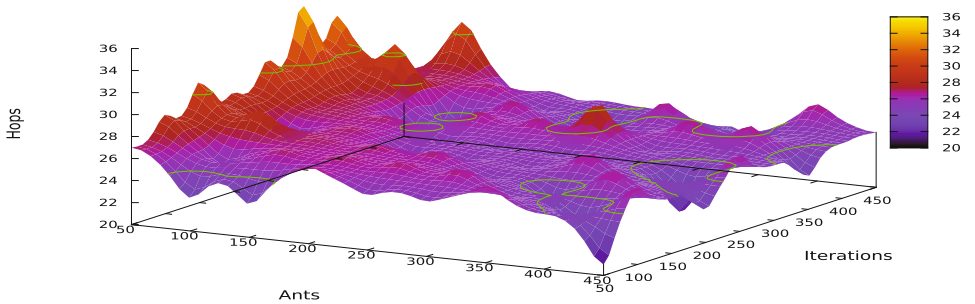
4 Experiments

In this section we provide several experiments used to evaluate our proposal for mitigating wildfire in natural ecosystems. In order to accomplish this evaluation, we have implemented Algorithm 1 using the simulation framework OMNeT++ 5.0 [1]. The implementation of Algorithm 1 and the GUI to execute experiments are available at http://antares.sip.ucm.es/tools/ants/index.html.

In order to analyse both the usability and accuracy of Algorithm 1, 3 different surfaces have been modelled. These surfaces have been configured with a



(a) Surface with 1 fire focus.

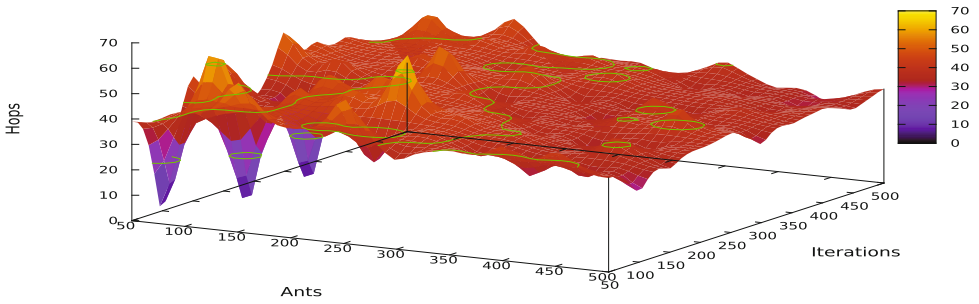


(b) Surface with 3 fire focuses.

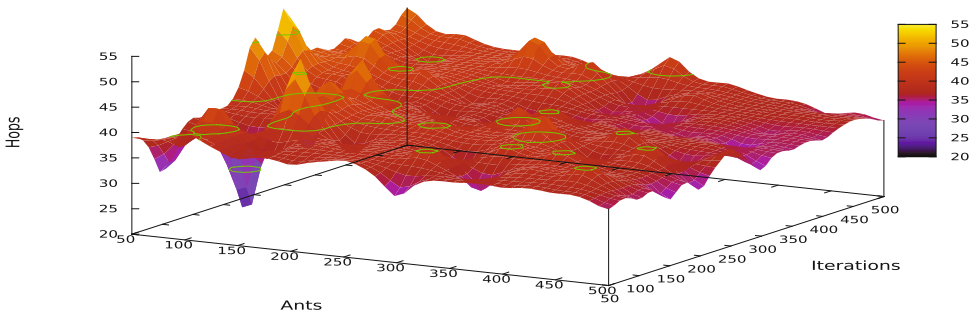
Fig. 1. Number of hops to mitigate wildfire in an surface of 25×25 regions

pre-defined number of fire focuses, which represents an active wildfire. In these experiments, we use 1 and 3 fire focuses for each surface. Also, we have used a range of values between 50 and 450 for both the number of ants and the number of iterations. Each region used to model these surfaces covers an area of $10m^2$. The proposed algorithm (see Algorithm 1) has been executed on these surfaces with the aim of finding the best path to mitigate the propagation of the existing fire focuses.

The first surface consists of a grid of 25×25 regions. The size of this surface is equivalent to almost 9 soccer fields. Figure 1(a) shows the results obtained by executing Algorithm 1 in this surface with 1 fire focus, while Fig. 1(b) shows the results for the same surface with 3 fire focuses. These charts depict the length of the shortest path to reach all the fire focuses from an initial point in the grid. In general, the length of the paths obtained when 1 fire focus is used ranges between 23 and 39. There is one exception when 50 ants and 150 iterations are used. In this case, the ants are not able to find the best path and they build a path containing a significant number of regions compared with the rest of the solutions. This fact is clearly reflected in the chart when 3 fire focuses are used. In this case, using a low number of ants has a direct impact on the quality of



(a) Surface with 1 fire focus.



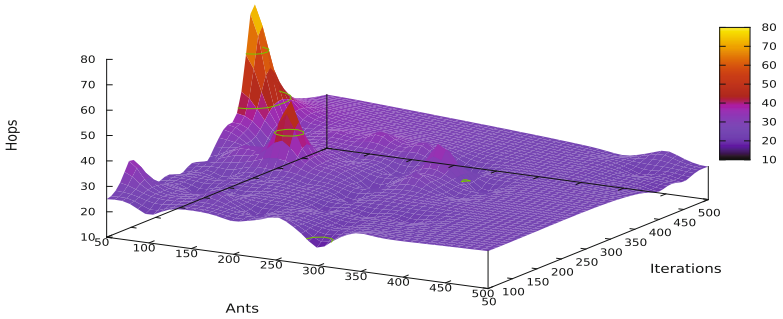
(b) Surface with 3 fire focuses.

Fig. 2. Number of hops to mitigate wildfire in an area of 50×50 regions

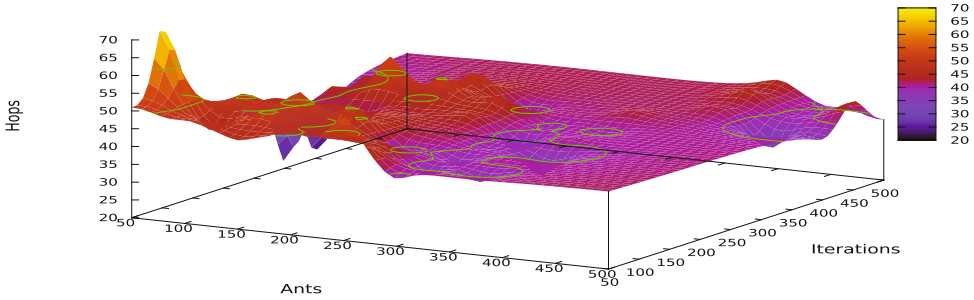
the obtained solution. Hence, increasing the number of ants slightly reduce the length of the obtained solution. However, the reduced dimensions of the terrain does not provide a significant reduction in the total length of the path when increasing the number of ants and iterations.

The second surface consists of a grid of 50×50 regions, which approximately represents an area 4 times larger than the gardens of Versalles. Figure 2(a) shows the results when 1 fire focus is used. Similarly to the previous surface, it can be seen a slight reduction in the length of the obtained path when the number of ants and the number of iterations increase. Moreover, due to the specific characteristic of the terrain, the algorithm executed with a low number of ants and iterations sometimes does not find a valid solution. This is reflected in the three peaks reaching zero. However, increasing the number of ants or the number of iterations seems to solve this problem, providing a suitable solution. Figure 2(b) shows the results when 3 fire focuses are used. In this case, the results obtained are similar to the ones obtained when 1 fire focus is used.

The last modelled surface consists of a grid of 100×100 regions, having a total extension nearly double of Vatican City. Figure 3(a) shows the results when



(a) Surface with 1 fire focus.



(b) Surface with 3 fire focuses.

Fig. 3. Number of hops to mitigate wildfire in an area of 100×100 regions

1 fire focus is used. In this case we can observe two interesting facts. First, there is a peak that represent a path much larger than the rest of the solutions when a low number of ants are used. Similarly than the first surface, in this case the ants were unable to find a suitable solution. Second, there are some variations in the length of the solution when the number of ants and iterations are below 350. This fact is mainly caused by the main characteristics of the terrain. However, when a high number of ants or iterations are used, the length of the solutions seems to be stabilized. Figure 3(a) shows the results when 3 fire focuses are used. Similarly, there are some variations in the length of the solutions. However, in this case, these variations occur when the number of ants used are less than 250. In the rest of the experiments, the solution seems to be stabilised. Using more than 450 ants and more than 450 iterations reflects a slight reduction in the length of the obtained path.

As a conclusion, increasing the number of ants executed in these modelled surfaces provides better solutions than using a low number of ants. This is mainly caused because increasing the number of executed ants also increases the probability of finding the best solution. When using a low number of ants, the solution can be improved by increasing the number of iterations. In some cases, using the maximum values for the number of ants and iterations may provide a slight improvement in the solution, which is more noticeable in large surfaces.

5 Conclusions and Future Work

In this paper we have developed a framework to model wildfire scenarios and simulate their evolution in natural ecosystems. First, we have modelled the main characteristics of the terrain for representing a wide variety of natural surfaces. Second, we provide a theoretical model that allows us to represent the fire in this environment. Finally, we have developed an algorithm, based on Ant Colony Optimization, to mitigate the wildfire spread in natural ecosystems.

In order to check both the usability and accuracy of this algorithm, we have carried out some experiments by modelling three different surfaces. Thus, the proposed algorithm were executed in each one the these surfaces. We can observe that increasing the number of ants provides better solutions than using a low number of ants. This means that using a high number of ants increase the probability of finding the best solution to reach all the fire focuses. Additionally, when using a low number of ants, the quality of the obtained solution can be improved by increasing the number of iterations.

Future work include modelling larger surfaces to analyse the scalability of the proposed algorithm. Also, we plan to extend the provided language to model a wider spectrum of natural ecosystems.

References

1. OMNeT++ 5.0 (2016). https://omnetpp.org/
2. Albini, F.A.: A model for fire spread in wildland fuels by-radiation. Combust. Sci. Technol. **42**(5–6), 229–258 (1985)

3. Alonso-Betanzos, A., Fontenla-Romero, O., Guijarro-Berdiñas, B., Hernández-Pereira, E., Andrade, M.I.P., Jiménez, E., Soto, J.L., Carballas, T.: An intelligent system for forest fire risk prediction and fire fighting management in Galicia. Expert Syst. Appl. **25**(4), 545–554 (2003)
4. Avesani, P., Perini, A., Ricci, F.: Combining CBR and constraints reasoning in planning forest fire fighting. In: 1st European Workshop on Case-based reasoning, pp. 235–239 (1993)
5. Beaufort, F.: Beaufort wind scale. British Rea-Admiral (1805)
6. Bowman, D., Balch, J., Artaxo, P., Bond, W., Carlsonand, J.M., Cochrane, M.A., D'Antonio, C.M., DeFries, R.S., Doyle, J., Harrison, S., Johnston, F., Keeley, J., Krawchuk, M., Kull, C., Marstond, J.B., Moritz, M., Prentice, I., Roos, C., Scott, A., Swetnam, T.W., van der Werf, G., Pyne, S.: Fire in the earth system. Science **324**(5926), 481–484 (2009)
7. Dale, L.: The true cost of wildfire in the Western US. Western Forestry Leadership Coalition (2009)
8. FAO (Food and Agriculture Organization of the United Nations). Global forest resources assessment 2010 (2010)
9. Giglio, L., Randerson, J.T., Werf, G.R.: Analysis of daily, monthly, and annual burned area using the fourth-generation global fire emissions database (gfed4). J. Geophys. Res.: Biogeosci. **118**(1), 317–328 (2013)
10. Karafyllidis, I., Thanailakis, A.: A model for predicting forest fire spreading using cellular automata. Ecol. Model. **99**(1), 87–97 (1997)
11. Mora, C., Tittensor, D.P., Adl, S., Simpson, A., Worm, B.: How many species are there on earth and in the ocean? PLoS Biol. **9**(8), e1001127 (2011)
12. Morvan, D., Dupuy, J.L.: Modeling of fire spread through a forest fuel bed using a multiphase formulation. Combust. Flame **127**(1–2), 1981–1994 (2001)
13. Nasi, R., Meijaard, E., Applegate, G., Moore, P.: Forest fire and biological diversity. Unasylva **53**(209) (2002)
14. R.C. Rothermel. A mathematical model for predicting fire spread in wildland fuels. Forest Services, (1972)
15. Sakr, G., Elhajjand, I.H., Mitriand, G., Wejinya, U.C.: Artificial intelligence for forest fire prediction. In: International Conference on Advanced Intelligent Mechatronics (AIM), pp. 1311–1316. IEEE (2010)
16. Weber, R.O.: Modelling fire spread through fuel beds. Prog. Energy Combust. Sci. **17**(1), 67–82 (1991)
17. Wiering, M.A., Dorigo, M.: Learning to control forest fires. In: 12th International Symposium on Computer Science for Environmental Protection, pp. 378–388. Metropolis Verlag (1998)

Truth is the ultimate power.
When the truth comes around
all the lies have to run and hide.

Ice cube

Say goodbye to yesterday

Non phixion